

Machine Learning

1-Les fondations du Machine Learning :

L'Homme a toujours eu des problèmes à résoudre pour pouvoir évoluer. Sauf que l'Homme est tombé sur des calculs de plus en plus complexes qui mettraient parfois des millions d'années avant d'être résolu, c'est pour ça que l'ordinateur a été inventé. L'ordinateur permet de résoudre des calculs complexes rapidement, mais il existe 2 *possibilités de nos jours* :

- Soit on connaît le calcul à résoudre et dans ce cas on le donne à l'ordinateur qui va le résoudre : **on appelle ça la programmation**
- Soit on ne connaît pas le calcul à résoudre et dans ce cas on ne peut donner de calcul à l'ordinateur, donc il faut que l'ordinateur apprenne par exemple à conduire, détecter des visages etc : on appelle ça : **le Machine Learning**

[Laisser la Machine apprendre à partir d'expériences]

Le Machine Learning consiste à laisser l'ordinateur apprendre quel calcul effectuer sans le programmer de façon explicite.

Définition de : **Arthur Samuel, inventeur du Machine Learning ayant développé un programme pouvant apprendre tout seul à jouer aux Dames en 1959**

[Une Machine apprend quand sa performance s'améliore]

Une machine apprend quand sa performance à faire une certaine tâche s'améliore avec des nouvelles expériences.

Définition de : **Tom Mitchell** en 1998 qui donna une définition plus moderne du Machine Learning.

Mais Comment Apprendre ?

On utilise des méthodes d'apprentissage qui sont fortement inspirées de la façon dont les humains apprennent à faire les choses. En apprentissage il existe :

- L'apprentissage supervisée (Supervised Learning)
- L'apprentissage non supervisée (Unsupervised Learning)
- L'apprentissage par renforcement (Reinforcement Learning)

L'apprentissage supervisée est l'apprentissage le plus utilisé en Machine Learning

L'Apprentissage Supervisé :

On va fournir à une machine beaucoup d'exemples qu'elle doit étudier.

Pour maîtriser cette apprentissage il faut connaître 4 notions:

- Le Dataset
- Le Modèle et ses Paramètres
- La Fonction Coût
- L'Algorithme d'apprentissage

Notion 1 : Le Dataset (apprendre à partir d'exemples)

On va fournir ici à la machine beaucoup d'exemples (x,y) dans le but de lui faire apprendre la relation qui relie x à y .

Exemple : Bonjour (en français) = Hello (en anglais)

On va compiler ces exemples (x,y) d'un domaine précis que l'on souhaite faire apprendre à la machine dans un tableau et ce tableau se nomme le Dataset (jeu de données).

Dans ce tableau :

- La variable **y** porte le nom de **target** (la cible). C'est à dire la valeur que l'on cherche à prédire.
- La variable **x** porte le nom de **feature** (caractéristique). Une caractéristique influence la valeur de y, en général on a beaucoup de caractéristiques ($x_1, x_2, x_3 \dots$) dans notre Dataset que l'on regroupe dans une matrice X.
(Plus on a de caractéristiques utiles à l'apprentissage plus notre modèle sera performant par la suite)

Attention : La construction d'un Dataset est une étape cruciale et importante pour après avoir un modèle intelligent et performant !

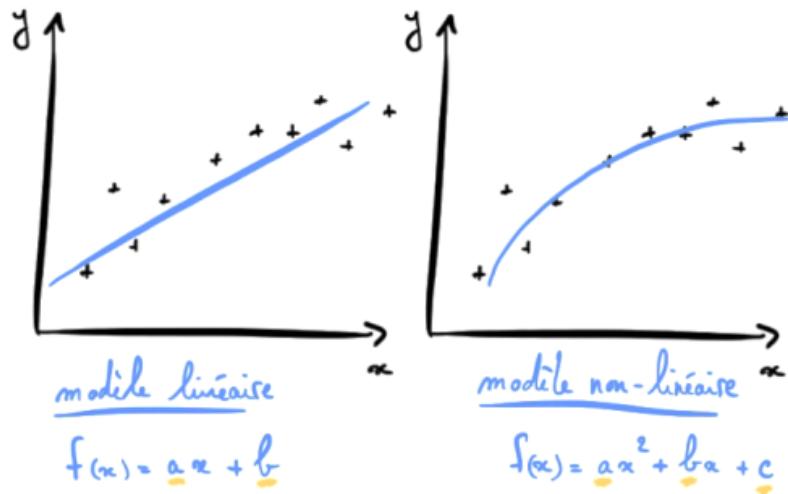
Exemple : c'est comme un bébé, si vous lui donnez de la nourriture saine alors il pourra grandir correctement et évoluer, par contre si vous lui donnez de la mauvaise nourriture sa croissance sera compromise. C'est pareil pour un modèle, si vous donnez des mauvaises données, votre modèle sera mauvais et inversement.

Ci-dessous, un Dataset qui regroupe des exemples d'appartements avec leur prix **y** ainsi que certaines de leurs caractéristiques (**features**).

Target y	Features		
	x_1	x_2	x_3
Prix	Surface m2	N chambres	Qualité
€ 313,000.00	124	3	1.5
€ 2,384,000.00	339	5	2.5
€ 342,000.00	179	3	2
€ 420,000.00	186	3	2.25
€ 550,000.00	180	4	2.5
€ 490,000.00	82	2	1
€ 335,000.00	125	2	2

Notion 2 : Développer un modèle à partir du Dataset

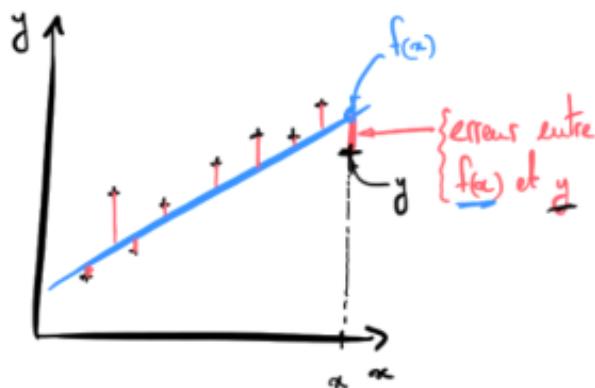
On peut choisir de développer un modèle linéaire ou un modèle non-linéaire selon les données et le besoin.



(on définit a, b, c etc comme étant les paramètres d'un modèle)

Notion 3 : Les erreurs de notre modèle – la fonction Coût

Un modèle nous retourne des erreurs par rapport à notre Dataset.
On appelle **Fonction Coût** l'ensemble de ces erreurs.



Fonction Coût = l'ensemble des erreurs .

Le plus souvent on prend la moyenne quadratique des erreurs.
(Mean Square Error)

Pour calculer la moyenne quadratique, on va d'abord prendre le carré de chaque valeur, puis faire la moyenne des carrés et enfin prendre la racine carré de la moyenne des carrés

Exemple :

$$2.5=6.25$$

$$3.2=10.24$$

$$2.7=7.29$$

$$3.8=14.44$$

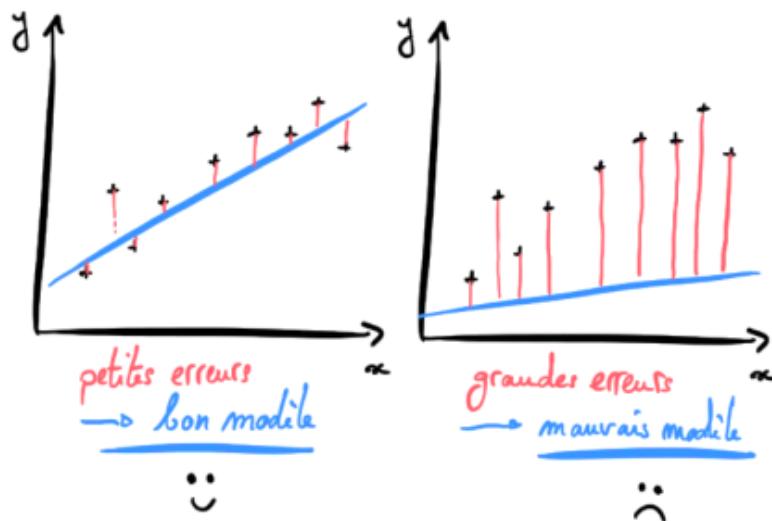
$$2.9=8.41$$

Moyenne des carrées =

$$6.25+10.24+7.29+14.44+8.41 / 5 = 46,63 / 5 = 9,326$$

Racine carré de la moyenne des carrées = $\sqrt{9,326} = 3,055$

Le but ici, c'est d'avoir un modèle qui nous donne de **petites erreurs**, donc une **petite Fonction Coût**.



Notion 4 : Apprendre, c'est minimiser la Fonction Coût

L'objectif central en Supervised Learning, c'est de trouver les **paramètres** du modèle qui minimisent la **Fonction Coût**.

Pour cela, on utilise l'algorithme d'apprentissage, l'exemple le plus courant étant l'algorithme de Gradient Descent (la descente de gradient).

Les applications du Supervised Learning :

Avec le Supervised Learning on peut développer des modèles pour résoudre 2 types de problèmes :

- Les problèmes de Régression
- Les problèmes de Classification

Dans les problèmes de régression, on cherche à prédire la valeur d'une variable **continue**, c-a-d une variable qui peut prendre une **infinité** de valeurs.

Exemple :

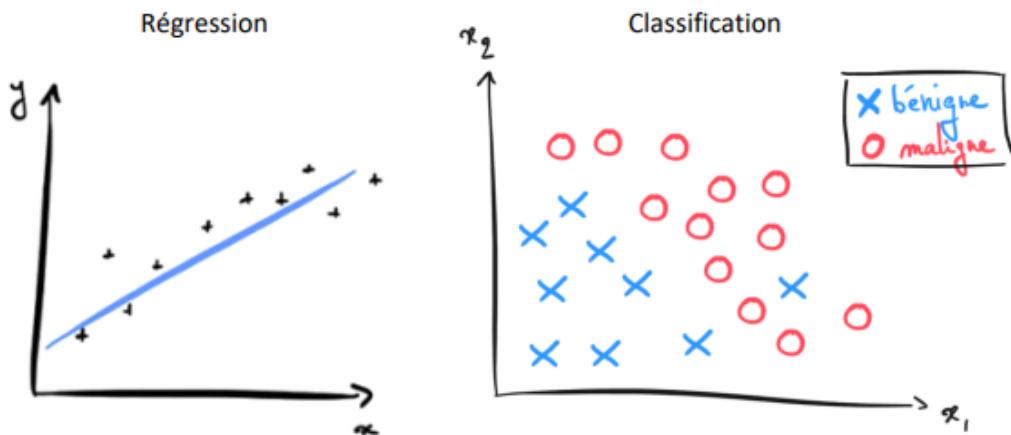
- 1) Prédire le prix d'un appartement (y) selon sa surface (x)
- 2) Prédire la quantité d'essence consommée (y) selon la distance parcourue (x)

Dans les problèmes de classification, on cherche à **classer** un objet dans différentes classes, c-a-d que l'on cherche à prédire la valeur d'une variable discrète (qui ne prend qu'un nombre **fini** de valeurs).

Exemple :

- 1) Prédire si un email est un spam ($y=1$) ou non ($y=0$) selon le nombre d'emails (x)
- 2) Prédire si une tumeur est maligne ($y=1$) ou bénigne ($y=0$) selon la taille de la tumeur (x_1) et l'âge du patient (x_2)

En classification on représente souvent les classes par des symboles, plutôt que par leur valeur numérique :



Info : Plus il y a de features disponibles et plus il y a de data (bonne) et plus il y a de chance que le modèle prenne de bonnes décisions dites '**intelligentes**' \Rightarrow c'est l'intelligence artificielle !

Autres méthodes d'apprentissage :

Il nous reste à voir, l'apprentissage non-supervisé et l'apprentissage par renforcement.

Il existe aussi l'apprentissage semi-supervisé, qui va combiner l'apprentissage supervisée et non supervisée.

Voici une brève introduction :

- L'apprentissage non-supervisé est une approche d'apprentissage automatique où un algorithme est utilisé pour trouver des modèles et des structures cachées dans un ensemble de données sans avoir besoin d'étiquettes ou de réponses préalables. C'est un apprentissage en pure autonomie.

- L'apprentissage par renforcement est utilisé principalement dans la robotique et s'inspire dans la façon dont les humains éduquent

leurs animaux de compagnies, en leur offrant une friandise quand ils font une bonnes actions et en les punissant quand ils en font une mauvaises.

4 Notions clefs en Machine Learning :

1-/ Le Dataset : tout démarre d'un Dataset qui contient nos données. En apprentissage supervisé, le Dataset contient les questions (x) et les réponses (y) au problème que la machine doit résoudre.

2-/ Le modèle et ses paramètres : à partir de ce Dataset on crée un modèle, qui n'est autre qu'une fonction mathématique. Les coefficients de cette fonction sont les paramètres du modèle.

3-/ La Fonction Coût : Lorsque l'on teste notre modèle sur le Dataset, celui-ci donne des erreurs. L'ensemble de ces erreurs, c'est ce qu'on appelle la Fonction Coût.

4-/ L'algorithme d'apprentissage : l'idée centrale dans le Machine Learning, c'est de laisser la machine trouver quels sont les paramètres de notre modèle qui minimisent notre Fonction Coût et donc intrinsèquement réduit les erreurs faites par notre modèle.

2-La Régression Linéaire :

Voici la recette à suivre pour réaliser son premier modèle de Machine Learning :

1- Récolter ses données :

Exemple :

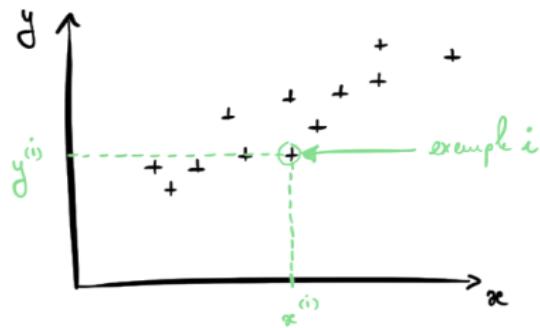
Imaginez que plusieurs agences immobilières vous aient fourni des données sur des appartements à vendre, notamment le prix de l'appartement (y) et la surface habitable (x). En Machine Learning, on dit que vous disposez de m exemples d'appartements.

On désigne :

$x^{(i)}$ la surface habitable de l'exemple i

$y^{(i)}$ le prix de l'exemple i

En visualisant votre **Dataset**, vous obtenez le nuage de points suivant :



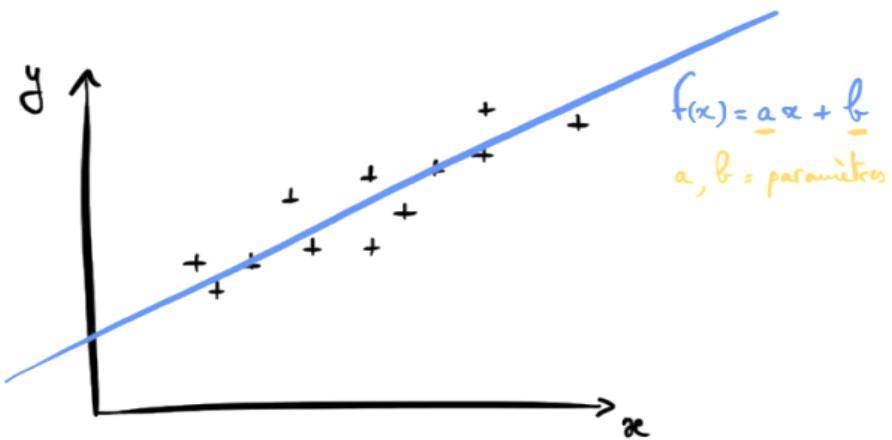
2- Créer un modèle linéaire :

A partir de nos données, on développe un modèle linéaire donc $f(x) = ax + b$

où a et b sont les paramètres du modèle.

Un bon modèle va donner de petites erreurs entre ses prédictions $f(x)$ et les exemples (y) du Dataset.

Nous ne connaissons pas les valeurs des paramètres a et b , ce sera le rôle de la machine de les trouver, de sorte à tracer un modèle qui s'insère bien dans notre nuage de point comme ci-dessous :



3- Définir la Fonction Coût :

Pour la régression linéaire, on utilise la **norme euclidienne** pour mesurer les erreurs entre $f(x)$ et (y) .

Rappel norme euclidienne :

La norme euclidienne est la distance euclidienne d'un vecteur à son origine.

Concrètement, voici la formule pour exprimer l'erreur i entre les **prix $y(i)$** et la prédiction faites en utilisant la **surface $x(i)$** :

$$\text{erreur}^{(i)} = (f(x^{(i)}) - y^{(i)})^2$$

Par exemple, imaginons que le 10ième exemple de notre Dataset soit un appartement de $x(10) = 80\text{m}^2$ dont le prix s'élève à $y(10) = 100\,000\text{€}$ et que notre modèle prédise un prix de $f(x(10))$ de $100\,002\text{€}$, l'erreur pour cette exemple est :

$$\text{erreur}^{(10)} = (f(x^{(10)}) - y^{(10)})^2$$

$$\text{erreur}^{(10)} = (100,002 - 100,000)^2$$

$$\text{erreur}^{(10)} = (2)^2$$

$$\text{erreur}^{(10)} = 4$$

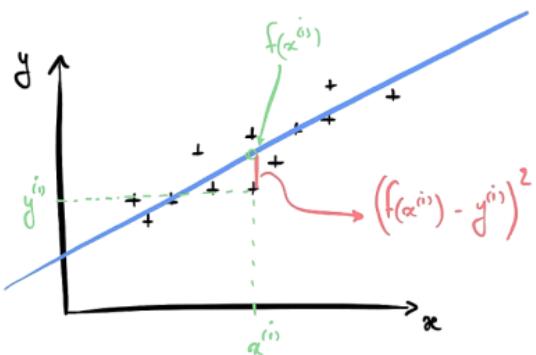
Chaque prédiction s'accompagne d'une erreur, on a donc **m** erreurs.

On définit la Fonction Coût $J(a,b)$ comme étant la moyenne de toutes les erreurs :

$$J(a, b) = \frac{1}{2m} \sum_{i=1}^m \text{erreur}^i$$

$$J(a, b) = \frac{1}{2m} \sum_{i=1}^m (f(x^{(i)}) - y^{(i)})^2$$

Et cette fonction se nomme l'erreur quadratique moyenne (Mean Squared Error)



$$J(a, b) = \frac{1}{2m} \sum_{i=1}^m (a x^{(i)} + b - y^{(i)})^2$$

Fonction Coût.

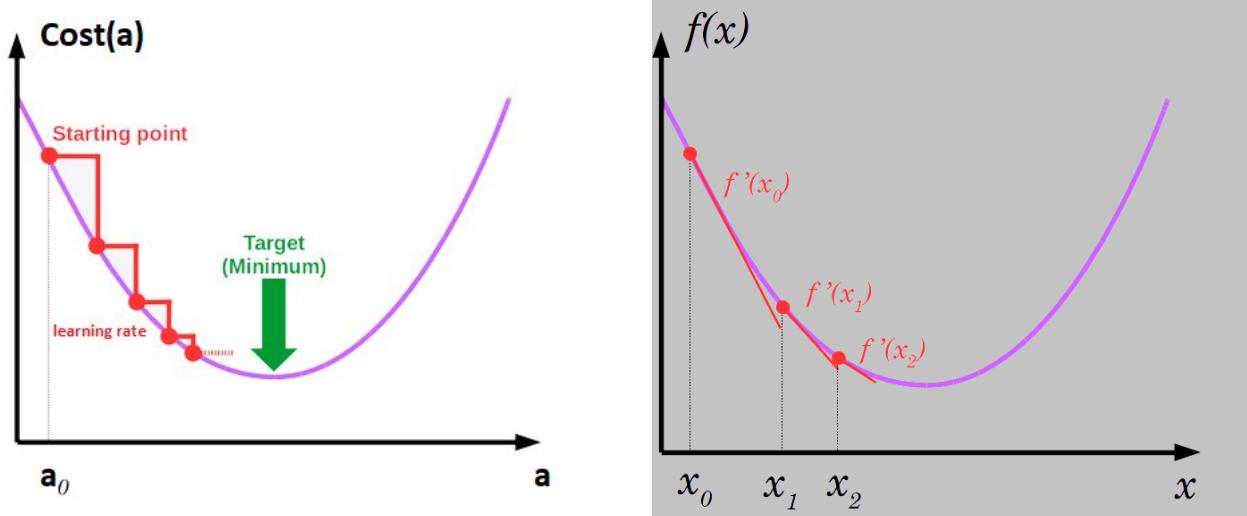
Et voici la Fonction Coût.

4- Trouver les paramètres qui minimisent la Fonction Coût :

Cette étape est la plus excitante, elle consiste à laisser la machine apprendre quels sont les meilleurs paramètres qui minimisent la Fonction Coût, c-a-d les paramètres qui nous donnent le meilleur modèle.

Pour trouver le minimum, on utilise un algorithme d'optimisation qui s'appelle la Descente de Gradient (Gradient Descent).

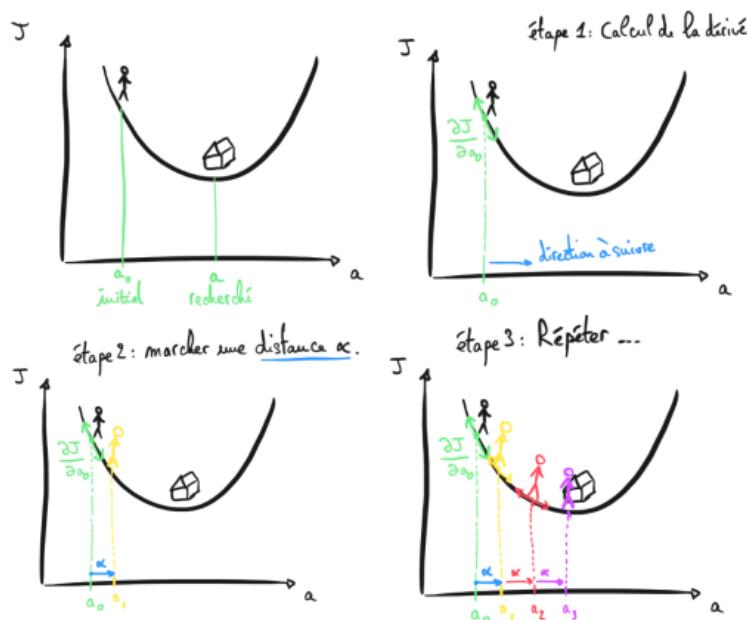
Comprendre le Gradient Descent :



Cet algorithme permet de trouver le minimum de la Fonction Coût, $J(a,b)$ en partant de coordonnées a et b aléatoires :

- 1- Calculer la pente de la Fonction Coût, c-a-d la dérivée de $J(a,b)$
- 2- Évoluer d'une certaine distance Alpha dans la direction de la pente la plus forte. Cela a pour résultat de modifier les paramètres a et b .
- 3- Recommencer les étapes 1 et 2 jusqu'à atteindre le minimum de $J(a,b)$

Pour illustrer l'algorithme, voici une illustration de recherche du paramètre idéal (ici pour a mais la même chose s'applique pour b)



Maintenant voyons **Comment utiliser l'algorithme de Gradient Descent :**

Notre objectif final c'est de trouver les paramètres a et b (ici) qui minimisent la Fonction Coût J(a,b)

Pour cela, on choisit au hasard a et b, puis nous allons utiliser en boucle la descente de gradient pour mettre à jour nos paramètres dans la direction de la Fonction Coût la plus faible.

Donc ici on va répéter en boucle :

$$a = a - \alpha \frac{\partial J(a, b)}{\partial a}$$

$$b = b - \alpha \frac{\partial J(a, b)}{\partial b}$$

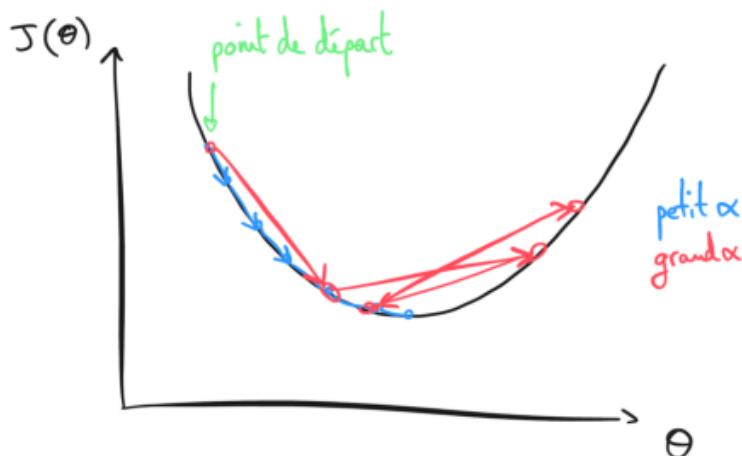
Explanation :

A chaque itération de cette boucle, les paramètres a et b sont mis à jour en soustrayant leur propre valeur à la valeur de la pente $\frac{\partial J(a, b)}{\partial \dots}$ multipliée par la distance à parcourir Alpha.

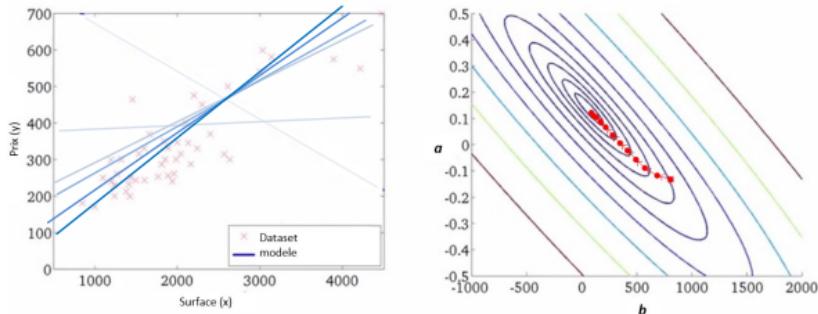
On appelle **Alpha** la vitesse d'apprentissage (**Learning Rate**).

Attention : si la vitesse est trop lente, le modèle peut mettre longtemps à être entraîné, mais si la vitesse est trop grande, alors la distance parcourue est trop longue et le modèle ne peut jamais converger vers le minimum.

Il est important de trouver un juste milieu :



Une fois cet algorithme programmé, c'est le moment le plus excitant en IA car on voit officiellement notre intelligence artificielle ***apprendre*** et dans notre cas par exemple, prédire le prix d'un appartement selon sa surface.



A partir de là, c'est la porte ouverte aux algorithmes qui automatisent et pratiquent des tâches, le même concept que l'on a vu va être appliqué majoritairement pour par exemple apprendre à un machine à reconnaître des visages sur une photo, prédire le cours de la bourse etc

MAIS, avant de voir la magie s'opérer il faut avoir préalablement calculé **les dérivés partielles de la Fonction Coût**.

Calcul des dérivés partielles :

Pour implémenter l'algorithme de Gradient Descent, il faut donc calculer les dérivées partielles de la Fonction Coût.

Rappel : La dérivée d'une fonction en un point nous donne la valeur de sa pente en ce point.

Fonction Coût :

$$J(a, b) = \frac{1}{2m} \sum_{i=1}^m (ax^{(i)} + b - y^{(i)})^2$$

Dérivée selon le paramètre a :

$$\frac{\partial J(a, b)}{\partial a} = \frac{1}{m} \sum_{i=1}^m (ax^{(i)} + b - y^{(i)}) \times x^{(i)}$$

Dérivée selon le paramètre b :

$$\frac{\partial J(a, b)}{\partial b} = \frac{1}{m} \sum_{i=1}^m (ax^{(i)} + b - y^{(i)})$$

Explication / indication :

$$(g \circ f)' = f' \times g' \circ f$$

Avec : $f = ax + b - y$ et $g = (f)^2$

En dérivant, le carré tombe et se simplifie avec la fraction $\frac{1}{2m}$ pour devenir $\frac{1}{m}$ et $x^{(i)}$ apparaît en facteur pour la dérivée par rapport à a .

Utilisation des matrices et vecteurs :

Dans la pratique, on exprime notre Dataset et nos paramètres sous forme **matricielle**, ce qui simplifie beaucoup les calculs. On créer ainsi un **vecteur** $\theta = \begin{pmatrix} a \\ b \end{pmatrix} \in \mathbb{R}^{n+1}$ qui contient tous les paramètres pour notre modèle, un **vecteur** $y \in \mathbb{R}^{m \times 1}$ et une **matrice** $X \in \mathbb{R}^{m \times n}$ qui inclut toutes les **features** n . Dans la régression linéaire, $n = 1$.

Au cas où vous seriez rouillé en algèbre : une matrice $\mathbb{R}^{m \times n}$, c'est comme un tableau avec m lignes et n colonnes.

Résumé des étapes pour développer un programme de Régression Linéaire :

1. Récolter des données (X, y) avec $X, y \in \mathbb{R}^{m \times 1}$
2. Donner à la machine un **modèle linéaire** $F(X) = X \cdot \theta$ où $\theta = \begin{pmatrix} a \\ b \end{pmatrix}$
3. Créer la **Fonction Coût** $J(\theta) = \frac{1}{2m} \sum (F(X) - y)^2$
4. Calculer le gradient et utiliser l'algorithme de **Gradient Descent**

Répéter en boucle:

$$\theta = \theta - \alpha \times \frac{\partial J(\theta)}{\partial \theta}$$

$$\text{Gradient: } \frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} X^T \cdot (F(X) - Y)$$

Précision et explication de l'équation finale du Gradient ici :

L'objectif est de trouver les valeurs de \vec{a} qui minimisent la fonction de coût $J(\vec{a}, b)$. Lorsque nous calculons le gradient de cette fonction de coût, nous voulons mesurer comment les paramètres \vec{a} et b influencent le changement de J . Cela implique de calculer les dérivées partielles de J par rapport à chaque composant de \vec{a} et à b .

Dans la formulation vectorielle, l'expression $\frac{1}{m} X^T (F(X) - Y)$ permet de regrouper ces dérivées partielles en une seule expression. X^T transpose la matrice X , ce qui nous permet d'obtenir les dérivées par rapport à chaque composant de \vec{a} en une seule opération matricielle. Ensuite, la multiplication $X^T (F(X) - Y)$ permet de combiner ces dérivées avec les erreurs entre les prédictions $F(X)$ et les valeurs cibles Y .

En somme, l'utilisation de X^T facilite le calcul de toutes les dérivées partielles en une seule étape, simplifiant ainsi la mise en œuvre de la descente du gradient pour la régression linéaire.

(en fait $X(T)$ ici c'est la transposition de la matrice)
(il ne faut pas oublier que X est une matrice ici)

Ce qui donnerait ici en Gradient pour J par rapport à a :

$$\frac{\partial J}{\partial a} = \frac{1}{m} \sum_{i=1}^m -(y_i - (ax_i + b))x_i$$

Et pour J par rapport à b :

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^m -(y_i - (ax_i + b))$$

Le Learning Rate (Alpha) prend le nom d'hyper-paramètre de par son influence sur la performance finale du modèle (trop grand le Gradient Descent ne converge pas, trop petit il prend trop de temps voir il arrive jamais au minimum).

3- Développer son Premier Programme de Régression Linéaire en Machine Learning :

On va maintenant commencer à programmer notre programme, il existe pour ça plusieurs étapes :

1- Importer les librairies : Numpy, Matplotlib, Sklearn (la fonction make_regression & SGDRegressor)

Explication :

- *Numpy \Rightarrow pour manipuler notre Dataset en tant que matrice
- *Matplotlib \Rightarrow pour visualiser nos données
- *La fonction make_regression de Sklearn \Rightarrow pour générer un nuage de points (ici on va simuler des données)
- *La fonction SGDRegressor (signifie Stochastic Gradient Descent Regressor) qui contient le calcul de la Fonction Coût, des gradients, de l'algorithme de minimisation etc en somme, tous les calculs et équations mathématiques du chapitre 2

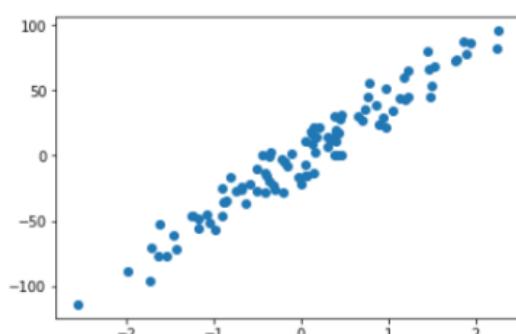
2- Créer un Dataset

On va ici générer des données (x,y) aléatoires.

On va utiliser la fonction make_regression pour ça. La fonction prend en arguments le nombre d'échantillons à générer, le nombre de variables et le bruit puis nous retourne 2 vecteurs x et y.

Pour maîtriser l'aléatoire on utilise « np.random.seed(0) », enfin pour visualiser nos données on utilise « plt.scatter(x,y) ».

Voici le résultat que vous devriez obtenir :



3- Développer le modèle et l'entraîner

Pour développer un modèle à la main il faut beaucoup de maths comme nous avons vu en chapitre 2 avec la Fonction Coût, les dérivés, les matrices, l'algo de Gradient Descent etc

Dans Sklearn tout est déjà implémenté.

Il suffit de définir une variable « model = SGDRegressor » puis entrer le nombre d'itérations du Gradient Descent que l'on veut et le Learning Rate.

Par exemple ici, entraînons le modèle sur 100 itérations avec un learning rate de 0,0001 :

```
« model = SGDRegressor(max_iter=100, eta0=0.0001) »
```

Une fois le modèle défini il faut l'entraîner, on va utiliser la fonction « fit ».

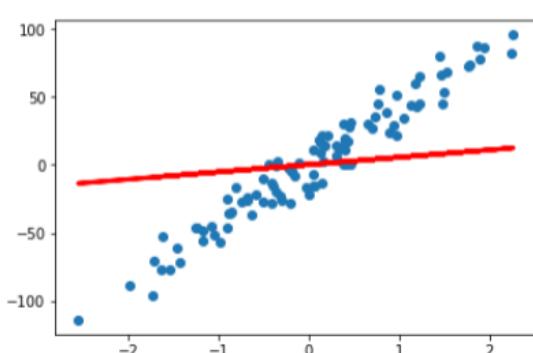
```
« model.fit(x,y) »
```

On peut observer la précision de notre modèle maintenant en utilisant la fonction « score » qui calcule le coefficient de détermination entre le modèle et les valeurs y de notre Dataset.

On peut également utiliser notre modèle pour faire des prédictions avec la fonction « predict » et tracer ses résultats avec « plt.plot » :

```
print('Coeff R2 =', model.score(x, y))
plt.scatter(x, y)
plt.plot(x, model.predict(x), c='red', lw = 3)
```

```
Coeff R2 = 0.22313211770520344
[<matplotlib.lines.Line2D at 0x1a111f552b0>]
```



Le modèle semble clairement mauvais. Mais c'est à cause que nous ne l'avons pas laissé s'entraîner assez longtemps et car le learning rate est trop faible, donc on va le refaire s'entraîner mais avec de meilleurs hyper-paramètres cette fois-ci :

En ML les valeurs qui fonctionnent bien pour la plupart des entraînements sont :

- Itérations = 1000
- Learning Rate = 0,001

Réessayons sur notre modèle :

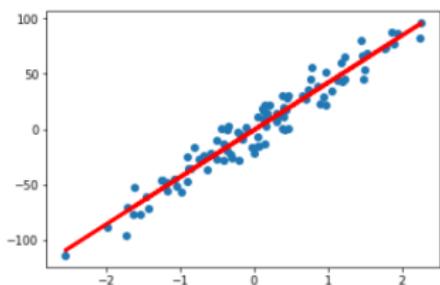
```
In [23]: model = SGDRegressor(max_iter=1000, eta0=0.001)
model.fit(x,y)

Out[23]: SGDRegressor(alpha=0.0001, average=False, early_stopping=False, epsilon=0.1,
eta0=0.001, fit_intercept=True, l1_ratio=0.15,
learning_rate='invscaling', loss='squared_loss', max_iter=1000,
n_iter=None, n_iter_no_change=5, penalty='l2', power_t=0.25,
random_state=None, shuffle=True, tol=None, validation_fraction=0.1,
verbose=0, warm_start=False)

In [24]: print('Coeff R2 =', model.score(x, y))
plt.scatter(x, y)
plt.plot(x, model.predict(x), c='red', lw = 3)

Coeff R2 = 0.9417290436914625

Out[24]: [<matplotlib.lines.Line2D at 0x1a111ffcb70>]
```



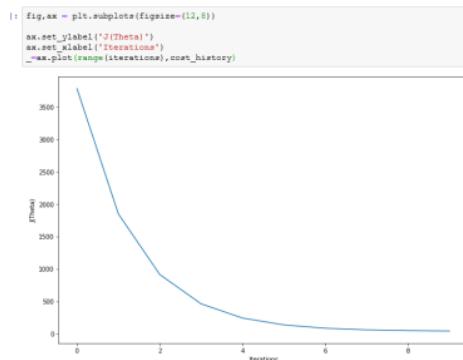
Parfait, voici notre premier modèle de ML entraîné et fonctionnel avec un coefficient R² de 94 % !!!

Ce qui donne un modèle avec de très bonnes prédictions !

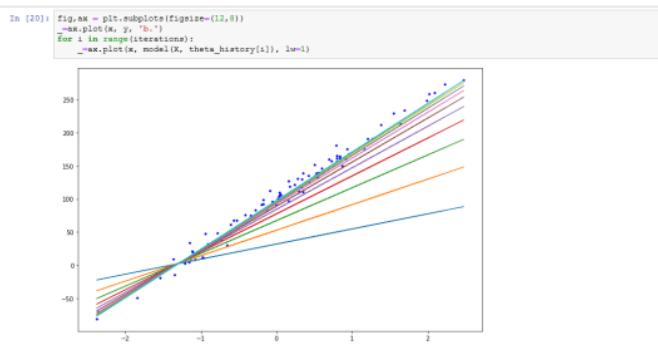
On va voir comment notre machine a appris les paramètres du modèle avec le Gradient Descent, pour cela il existe ce qu'on appelle **les courbes d'apprentissage**.

En ML on appelle courbe d'apprentissage (Learning Curves) les courbes qui montrent l'évolution de la Fonction Coût au fil des itérations du Gradient Descent.

Si notre modèle apprend alors la courbe doit diminuer au fil du temps :



A chaque itération, le modèle s'améliore pour donner la droite ci-dessous.



Et on a notre modèle, mais voici le code à la main si on devait faire tout nous même sans utiliser les fonctions pré-faites de Sklearn :

Importer les libraires :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression
```

Générer un Dataset aléatoire

```
np.random.seed(4)
n = 1
m = 100

x, y = make_regression(n_samples=m, n_features=n, noise=10)
y = y + 100
plt.scatter(x, y)
y = y.reshape(y.shape[0], 1)

#ajouter le Bias a X
X = np.hstack((np.ones(x.shape), x))
X.shape
```

Définir sous forme matricielle le modèle, la Fonction Coût et le gradient.
On définit θ le vecteur qui contient les paramètres a et b .

$$F = X \cdot \theta$$

$$J(\theta) = \frac{1}{2m} \sum (X \cdot \theta - y)^2$$

$$\text{Grad}(\theta) = \frac{1}{m} X^T \cdot (X \cdot \theta - y)$$

```
#definir la fonction modele
def model(X, theta):
    # x shape: (m, n)
    # theta shape: (n,1)
    return X.dot(theta) #shape: (m, 1)

#definir la fonction cout
def cost_function(X, y, theta):
    m = len(y)
    J = 1/(2*m) * np.sum((model(X, theta) - y)**2)
    return J

#definir la fonction gradient
def gradient(X, y, theta):
    return 1/m * X.T.dot((X.dot(theta) - y))
```

On définit la fonction Gradient Descent avec une boucle `for` :

For all itérations :

$$\theta = \theta - \alpha \text{Grad}(\theta)$$

```
#algorithme de Gradient Descent
def gradient_descent(X, y, theta, learning_rate = 0.001, iterations = 1000):
    m = len(y)
    cost_history = np.zeros(iterations)
    theta_history = np.zeros((iterations, 2))

    for i in range(0, iterations):
        prediction = model(X, theta)
        theta = theta - learning_rate * gradient(X, y, theta)
        cost_history[i] = cost_function(X, y, theta)
        theta_history[i,:] = theta.T

    return theta, cost_history, theta_history
```

On passe à l'entraînement du modèle, puis on visualise les résultats.

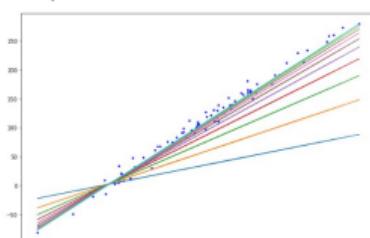
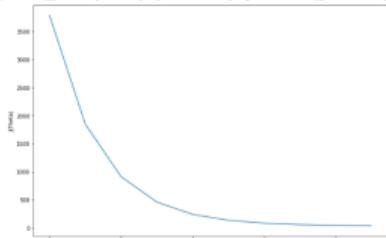
```
# utilisation de l'algorithme
np.random.seed(0)
theta = np.random.randn(2, 1)

iterations = 10
learning_rate = 0.3
theta, cost_history, theta_history = gradient_descent(X, y, theta, learning_rate=learning_rate, iterations = iterations)

#visualisation des courbes d'apprentissage
fig,ax = plt.subplots(figsize=(12,8))
```

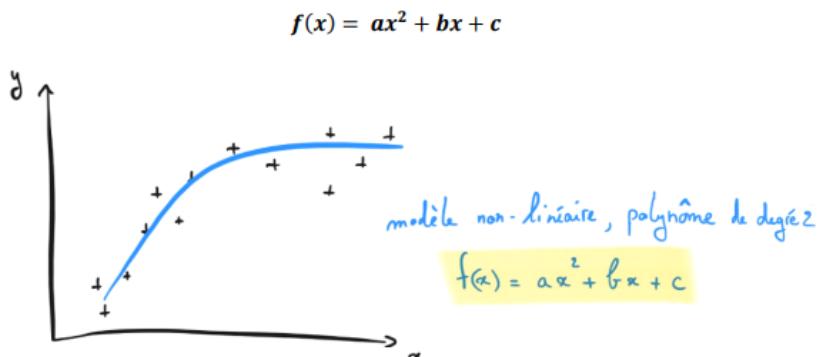
```
ax.set_ylabel('J(Theta)')
ax.set_xlabel('Iterations')
_=ax.plot(range(iterations),cost_history)

# visualisation du modèle au cours de son apprentissage
fig,ax = plt.subplots(figsize=(12,8))
_=ax.plot(x, y, 'b.')
for i in range(iterations):
   _=ax.plot(x, model(X, theta_history[i]), lw=1)
```



Régression polynomiale à plusieurs variables :

Pour un nuage de point comme ci-dessous il semble judicieux de développer un modèle polynomial de degré 2.



Ce modèle plus complexe va engendrer des calculs algébriques plus intenses notamment pour le calcul des dérivées, mais grâce à des bibliothèques comme Sklearn c'est simple.

Le code qu'on a écrit pour la régression linéaire peut être utilisé pour des problèmes plus complexes, il suffit de générer des variables polynomiales dans notre Dataset en utilisant la fonction « PolynomialFeatures » présente dans Sklearn :
« from sklearn.preprocessing import PolynomialFeatures »

Grâce au calcul matriciel (présent dans Numpy et Sklearn) la machine peut intégrer ces nouvelles variables polynomiales sans changer son calcul !

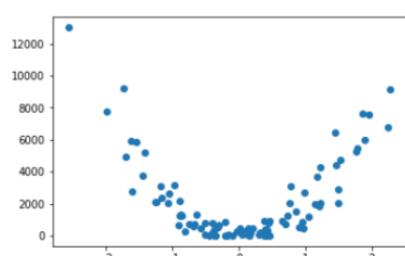
Voyons un exemple en ajoutant une variable polynomiale de degré 2 pour forcer la machine à développer un modèle qui aura une allure parabolique de y en fonction de x :

```
np.random.seed(0)

# création du Dataset
x, y = make_regression(n_samples=100, n_features=1, noise=10)
y = y**2 # y ne varie plus linéairement selon x !

# On ajoute des variables polynomiales dans notre dataset
poly_features = PolynomialFeatures(degree=2, include_bias=False)
x = poly_features.fit_transform(x)

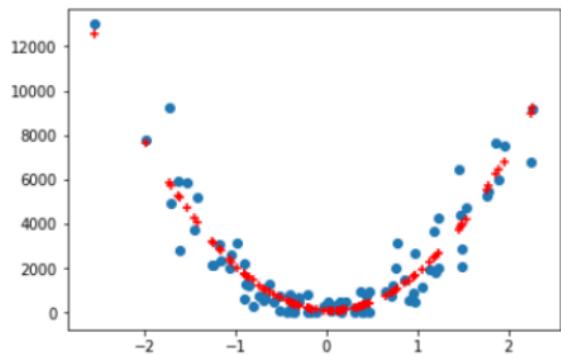
plt.scatter(x[:,0], y)
x.shape # la dimension de x: 100 lignes et 2 colonnes
```



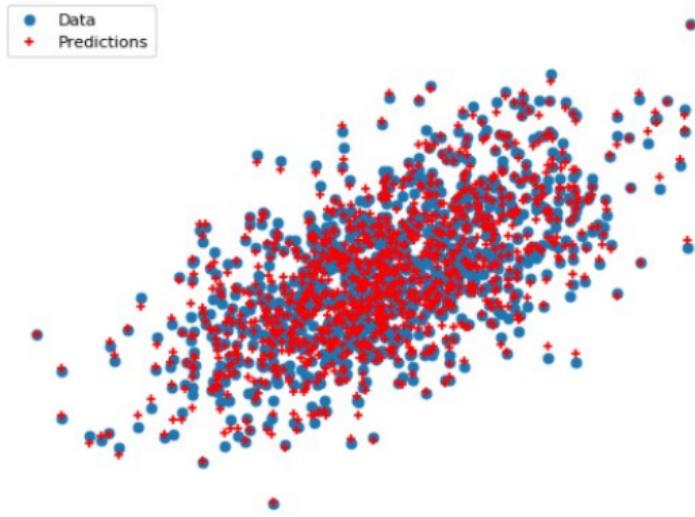
```
# On entraîne le modèle comme avant ! rien ne change !
model = SGDRegressor(max_iter=1000, eta0=0.001)
model.fit(x,y)
print('Coeff R2 =', model.score(x, y))

plt.scatter(x[:,0], y, marker='o')
plt.scatter(x[:,0], model.predict(x), c='red', marker='+')
```

Coeff R2 = 0.8940617695872648
<matplotlib.collections.PathCollection at 0x2110370f470>



Avec la fonction « PolynomialFeatures » on peut ainsi développer des modèles bien plus complexes et capable de prédire des résultats sur des milliers de dimensions, comme par exemple :



En résumé :

Avec Sklearn il suffit d'écrire quelques lignes pour développer des modèles de régression linéaire et polynomiale.

On a les fonctions principales suivantes :

- model = SGDRegressor(nb_itérations, learning_rate)
- model.fit(x, y) : pour entraîner votre modèle.
- model.score(x, y) : pour évaluer votre modèle.
- model.predict(x) : pour générer des prédictions.

4- Régression Logistique et Algorithme de Classification :

Les problèmes de Classification

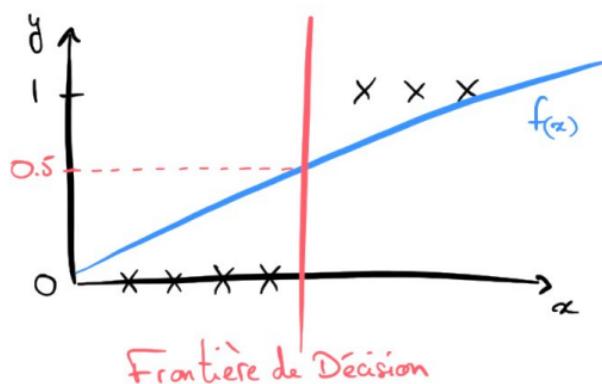
Cela consiste à classer un email en tant que spam ou non par exemple.

Dans ce genre de problème on aura un Dataset contenant une variable target y pouvant prendre **2 valeurs différentes** par exemple 0 ou 1.

- si $y=0$ alors email n'est pas un spam
- si $y=1$ alors l'email est un spam

On dit également que l'on a 2 classes, c'est une classification binaire.

Pour ces problèmes on ajoute une **frontière de décision** qui permet de classer un email par exemple dans la classe 0 ou 1.

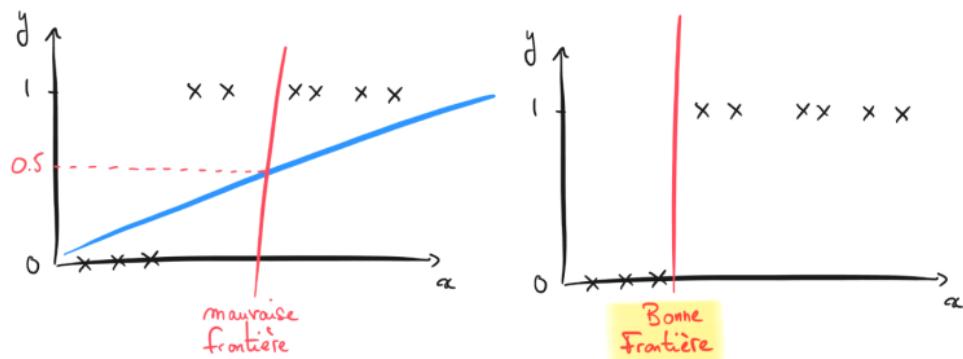


$$\begin{cases} \text{si } f(x) \leq 0.5 \text{ alors } y = 0 \\ \text{si } f(x) > 0.5 \text{ alors } y = 1 \end{cases}$$

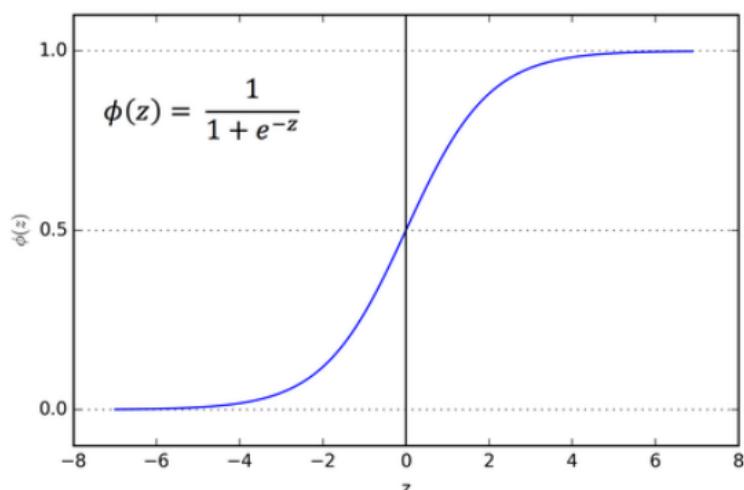
Le modèle de Régression Logistique :

Pour des problèmes de classification binaire, un modèle linéaire $F=X\cdot 0$ ne convient pas.
(ici le 0 c'est Theta)

Voyons le résultat qu'on obtient avec un tel modèle :



On développe alors une nouvelle fonction pour les problèmes de classification binaire, c'est la **fonction logistique** (aussi appelé fonction sigmoïde). Cette fonction a la particularité d'être toujours comprise entre 0 et 1.

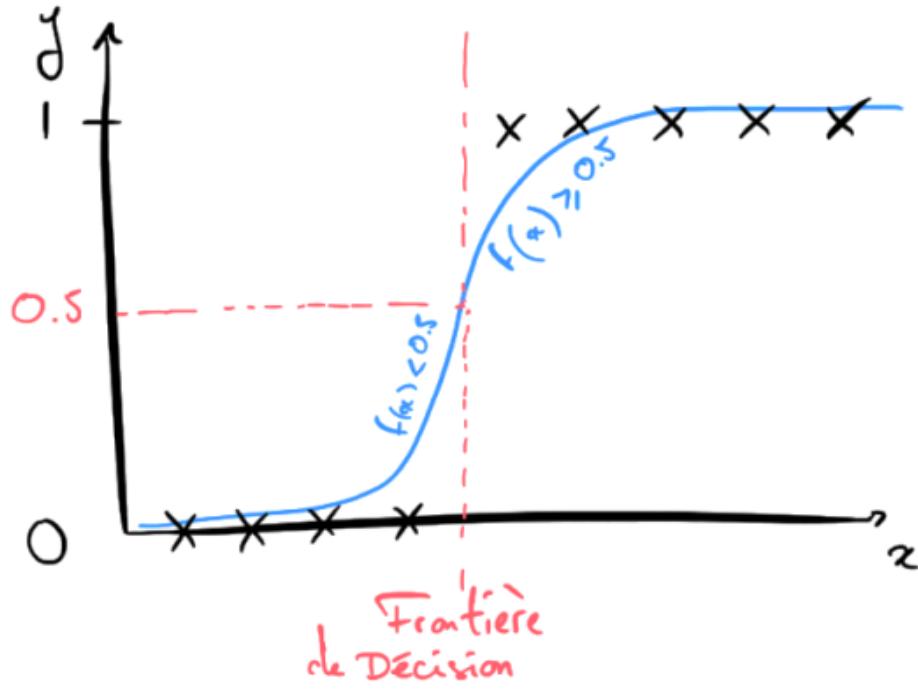


Pour coller la fonction logistique sur un Dataset(X, y) on y fait passer le produit matriciel $X \cdot \theta$ (ici θ = Theta) ce qui nous donne le modèle de Logistic Regression :

$$\sigma(X \cdot \theta) = \frac{1}{1 + e^{-X \cdot \theta}}$$

A partir de cette fonction, il est possible de définir une **frontière de décision**. Typiquement, on définit un seuil à 0.5 :

$$\begin{cases} y = 0 \text{ si } \sigma(X \cdot \theta) < 0.5 \\ y = 1 \text{ si } \sigma(X \cdot \theta) \geq 0.5 \end{cases}$$



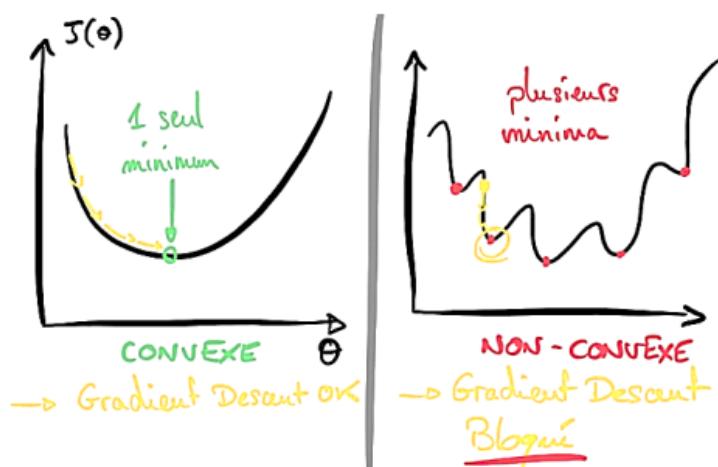
Fonction Coût associé à la Régression Logistique :

Rappel Fonction Coût pour régression linéaire :

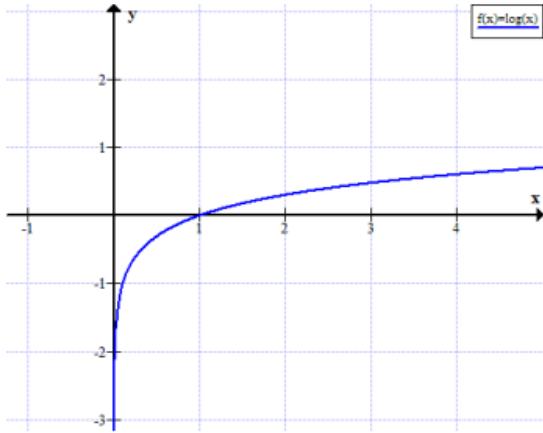
$$J(\theta) = \frac{1}{2m} \sum (X \cdot \theta - Y)^2$$

La Fonction Coût pour la régression linéaire nous donnait une courbe convexe (qui présente un unique minima). C'est ce qui fait que l'algorithme de Gradient Descent fonctionne.

Or, utiliser cette fonction pour le modèle Logistique ne donnera pas de courbe convexe (dû à la non-linéarité) et l'algorithme de Gradient Descent se bloquera en un minima local et non global car il va se bloquer au premier minima rencontré.



Il faut alors développer une **Fonction Coût spécialement pour la régression logistique**. On va alors utiliser la fonction **logarithme** pour transformer la fonction sigma en **fonction convexe** en séparant les cas où $y=1$ des cas où $y=0$:



Fonction Coût dans les cas où $y=1$:

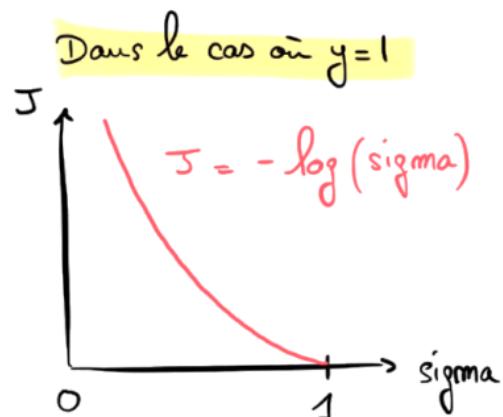
Voici la Fonction Coût que l'on utilise dans les cas où $y=1$:

$$J(\theta) = -\log(\sigma(X \cdot \theta))$$

Explications :

Si notre modèle prédit $\sigma(x) = 0$ alors que $y = 1$, on doit pénaliser la machine par une grande erreur (un grand coût). La fonction logarithme permet de tracer cette courbe avec une propriété convexe, ce qui poussera le Gradient Descent à trouver un paramètre Theta pour un coût qui tend vers 0.

De plus, quoi de mieux qu'une fonction logarithme qui va tendre vers l'infinie, qui permettra en faisant un $-\log$ d'avoir une forte pénalité qui tend vers l'infini, si, pour notre exemple actuel, il se rapproche de 0 au lieu d'aller à 1.



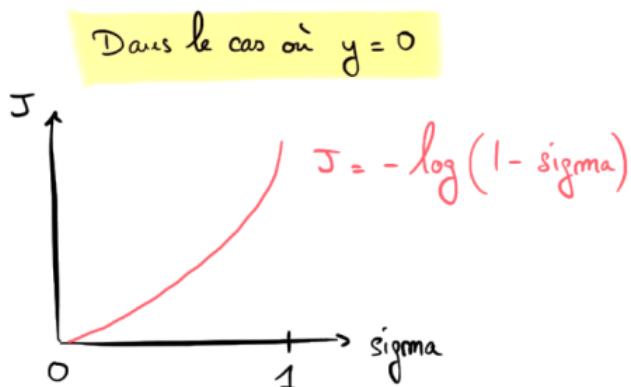
Fonction Coût dans les cas où $y = 0$:

Cette fois-ci la Fonction Coût devient :

$$J(\theta) = -\log(1 - \sigma(X.\theta))$$

Explications :

Si notre modèle prédit $\sigma(x) = 1$ alors que $y = 0$, on doit pénaliser la machine par une grande erreur (un grand coût). Cette fois $-\log(1-0)$ donne la même courbe mais est inversée sur l'axe vertical.



Fonction Coût Complète :

Pour écrire la Fonction Coût en une seule équation, on utilise l'astuce de séparer les cas $y=0$ et $y=1$ avec une annulation :

$$J(\theta) = \frac{-1}{m} \sum y \times \log(\sigma(X.\theta)) + (1 - y) \times \log(1 - \sigma(X.\theta))$$

Dans le cas où $y = 0$, il nous reste :

$$J(\theta) = \frac{-1}{m} \sum 0 \times \log(\sigma(X.\theta)) + 1 \times \log(1 - \sigma(X.\theta))$$

Et dans le cas où $y = 1$

$$J(\theta) = \frac{-1}{m} \sum 1 \times \log(\sigma(X.\theta)) + 0 \times \log(1 - \sigma(X.\theta))$$

Gradient Descent pour la Régression Logistique :

L'algorithme de Gradient Descent s'applique exactement de la même manière que pour la régression linéaire. En plus, la dérivée de la Fonction Coût reste la même, on a :

$$\text{Gradient: } \frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} \sum (\sigma(X \cdot \theta) - y) \cdot X$$

$$\text{Gradient Descent: } \theta = \theta - \alpha \times \frac{\partial J(\theta)}{\partial \theta}$$

Résumé de la Régression Logistique :

$$\text{Modèle: } \sigma(X \cdot \theta) = \frac{1}{1 + e^{-X \cdot \theta}}$$

$$\text{Fonction Coût: } J(\theta) = -\frac{1}{m} \sum y \times \log(\sigma(X \cdot \theta)) + (1 - y) \times \log(1 - \sigma(X \cdot \theta))$$

$$\text{Gradient: } \frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} X^T \cdot (\sigma(X \cdot \theta) - y)$$

$$\text{Gradient Descent: } \theta = \theta - \alpha \times \frac{\partial J(\theta)}{\partial \theta}$$

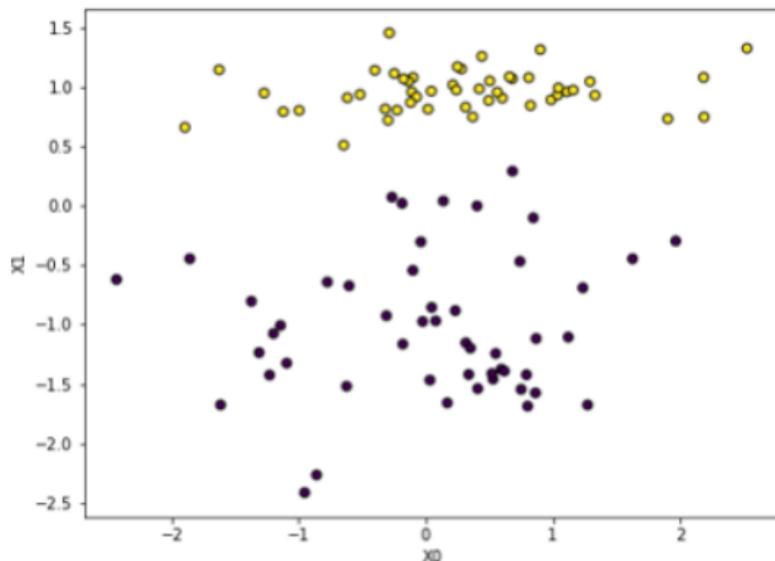
Développer un programme de Classification Binaire :

Comme précédemment, on va générer des données aléatoires mais cette fois-ci avec la fonction « make_classification ».

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.linear_model import SGDClassifier

# Génération de données aléatoires: 100 exemples, 2 classes, 2 features x0 et x1
np.random.seed(1)
X, y = make_classification(n_samples=100, n_features=2, n_redundant=0, n_informative=1,
                           n_clusters_per_class=1)

# Visualisation des données
plt.figure(num=None, figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], marker='o', c=y, edgecolors='k')
plt.xlabel('X0')
plt.ylabel('X1')
x.shape
```



Ensuite, on doit créer un modèle en utilisant « SGDClassifier ».

```
# Génération d'un modèle en utilisant la fonction cout 'log' pour Logistic Regression
model = SGDClassifier(max_iter=1000, eta0=0.001, loss='log')

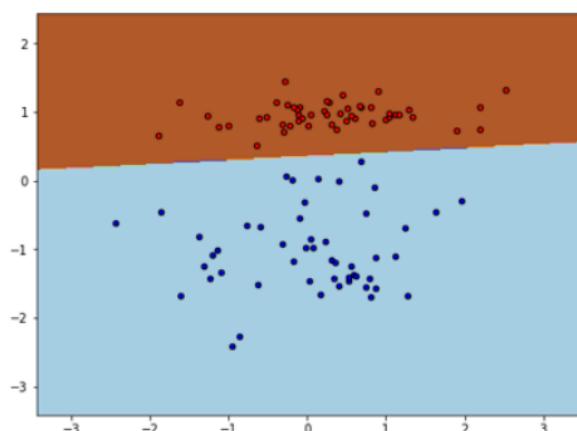
model.fit(X, y)
print('score:', model.score(x, y))
```

Une fois le modèle entraîné, on peut afficher sa frontière de décision :

```
# Visualisation des données
h = .02
colors = "bry"
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                      np.arange(y_min, y_max, h))

Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.axis('tight')

for i, color in zip(model.classes_, colors):
    idx = np.where(y == i)
    plt.scatter(X[idx, 0], X[idx, 1], c=color, cmap=plt.cm.Paired, edgecolor='black', s=20)
```



L'Algorithme de Nearest Neighbour : (en français : le voisin le plus proche)

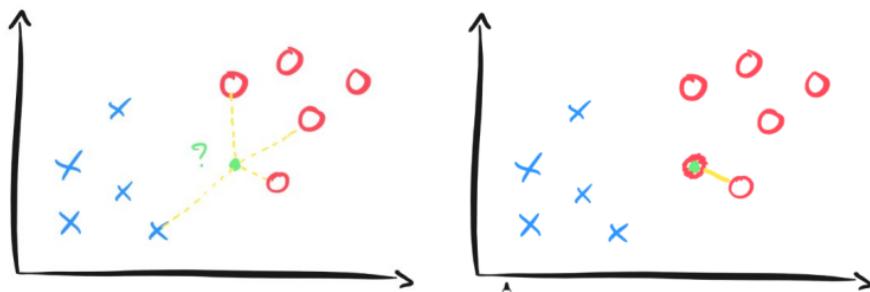
Il permet de résoudre des problèmes de **classification à plusieurs classes** de façon simple et très efficace.

Quand on fait une nouvelle prédiction, l'algorithme trouve dans notre Dataset l'exemple le plus proche par rapport aux conditions dans lesquelles on est.

K-Nearest Neighbour (K-NN) :

La distance la plus courte.

Regardons le nuage de point qui suit.



Quel est l'exemple le plus proche du point vert ? C'est un exemple de classe rouge.

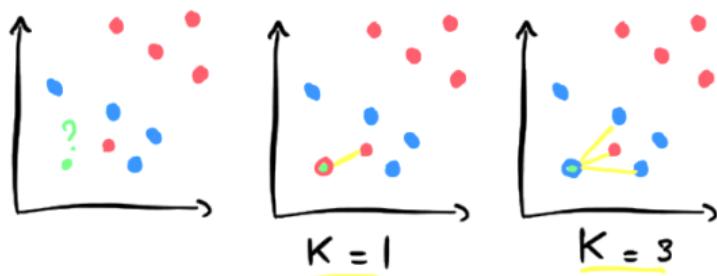
L'algorithme de Nearest Neighbour calcule ainsi la **distance** entre le point vert et les autres points du Dataset et associe le point vert à la classe dont l'exemple est le plus proche en terme de distance.

Typiquement, on utilise la **distance euclidienne** (droite directe entre 2 points) mais d'autres métriques sont parfois plus utiles, comme la distance de **Manhattan** ou bien la distance **cosinus**. (vous pouvez voir sur internet ce que c'est pour comprendre, mais ici Sklearn l'implémente pour nous)

Le nombre de voisin K

Pour limiter les problèmes liés au bruit (ce qu'on appelle l'**Over fitting**) on peut demander à l'algorithme de trouver les K voisins les plus proches du point vert.

Cela améliore la qualité du modèle car il devient moins sensible aux impuretés et cas particuliers qui viendraient empêcher la bonne **généralisation**.



Développement d'un programme de vision par ordinateur avec K-NN :

On peut charger depuis Sklearn un Dataset spécialement pour (Sklearn contient des Datasets de base).

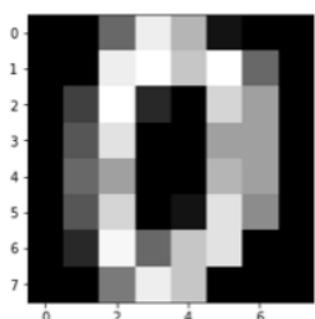
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.neighbors import KNeighborsClassifier
```

```
# importons une base de données de chiffre
digits = load_digits()

X = digits.data
y = digits.target

print('dimension de X:', X.shape)
```

```
dimension de X: (1797, 64)
<matplotlib.image.AxesImage at 0x1a34df0fa90>
```



Avec le code ci-dessus, on voit un exemple de chiffre présent dans le Dataset, c'est 0 ici.

On apprend que la Dataset comprend 1797 exemples, c-a-d 1797 images, et que chaque exemple contient 64 features.

Il s'agit donc de chacun des 64 pixels qui forment les images, 1 pixel = 1 feature ici.

Quand on soumet un nouveau chiffre à la machine, l'algorithme K-NN trouve l'exemple du Dataset qui ressemble le plus à notre chiffre, basé sur le voisin le plus proche pour la valeur de chaque pixel.

L'étape suivant consiste à entraîner le modèle de Nearest Neighbour.

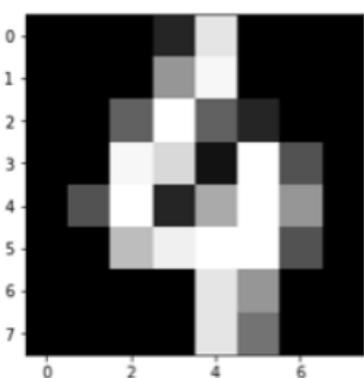
```
# visualisons un de ces chiffres
plt.imshow(digits['images'][0], cmap = 'Greys_r')

# Entrainement du modèle
model = KNeighborsClassifier()
model.fit(X, y)
model.score(X, y)
```

Maintenant, testons sur une image au hasard du Dataset, prenons la 100ième image qui est un 4 par exemple et voyons si la machine arrive à identifier le chiffre :

```
#Test du modèle
test = digits['images'][100].reshape(1, -1)
plt.imshow(digits['images'][100], cmap = 'Greys_r')
model.predict(test)

array([4])
```



La machine a clairement reconnu un 4 !

Bilan du chapitre :

On a vu 2 algorithmes très populaires pour les problèmes de Classification :

- La Régression Logistique avec Gradient Descent
- Le K-Nearest Neighbour

La fonction Logistique est une fonction **IMPORTANTE** dans l'histoire du ML car c'est **elle que l'on trouve au cœur des neurones des fameux Réseaux de Neurones Artificiels.**

5- Les Réseaux de Neurones :

On passe ici de Machine Learning à du Deep Learning (apprentissage profond).

Les réseaux de neurones sont des modèles bien plus complexes que tous les autres algorithmes de Machine Learning classique.

Ils représentent des fonctions mathématiques avec des millions de coefficients (les paramètres).

Alors que par exemple en régression linéaire on en a vu 2, a et b, ici un réseau en a des millions !

Avec une telle puissance, il est possible d'entraîner la machine sur des tâches bien plus avancées :

- La reconnaissance d'objets et faciale
- L'analyse de sentiments
- L'analyse du langage naturel
- La création artistique
- etc

Cependant, développer une fonction aussi complexe à un coût.
Pour y parvenir il faut souvent fournir :

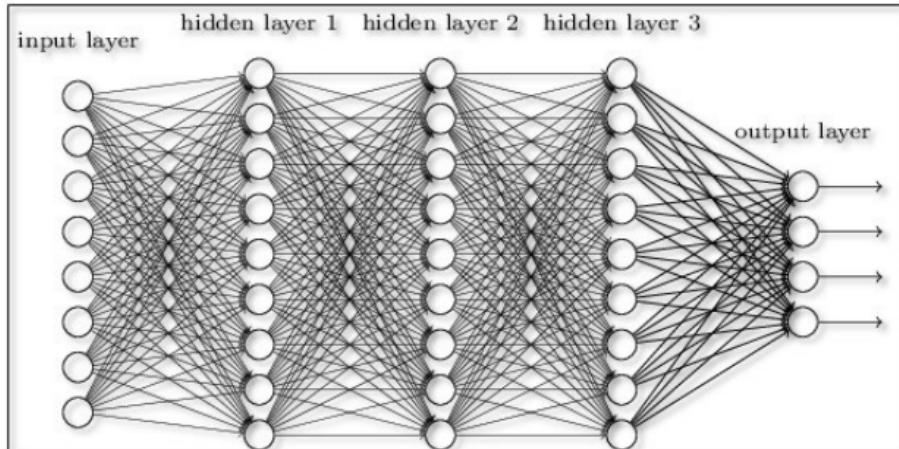
- Un Dataset beaucoup plus grand (des millions de données)
- Un temps d'apprentissage plus long (parfois des jours / semaines)
- Une plus grande puissance de calcul (et donc une infrastructure permettant d'encaisser la puissance requise)

Pour dépasser ces challenges, les chercheurs ont développés des **variantes** du Gradient Descent ainsi que d'autres techniques pour calculer plus rapidement les dérivées sur des millions de données.
Parmi ces solutions on trouve :

- Mini-Batch Gradient Descent : C'est une technique pour laquelle le Dataset est fragmenté en petits lots pour simplifier le calcul du gradient à chaque itérations
- Batch Normalization : Cela met à la même échelle toutes les variables d'entrée et de sortie internes au Réseau de neurones pour éviter d'avoir des calculs de gradients extrêmes.
- Distributed Deep Learning : Utilisation du Cloud pour diviser le travail et le confier à plusieurs machines.

Comprendre les Réseaux de Neurones :

Voici à quoi ressemble un réseau de neurones :



On a un niveau d'entrées (**input layer**) à gauche, un niveau de sorties (**output layer**) à droite et plusieurs niveaux cachés entre deux.

Les petits ronds sont appelés les **neurones** et représentent des **fonctions d'activations**. Pour un réseau de neurone basique, la **fonction Logistique** est utilisé comme fonction d'activation. C'est pour cela que nous l'avons vu précédemment.

Réseau de Neurone à 1 Neurone : Le Perceptron

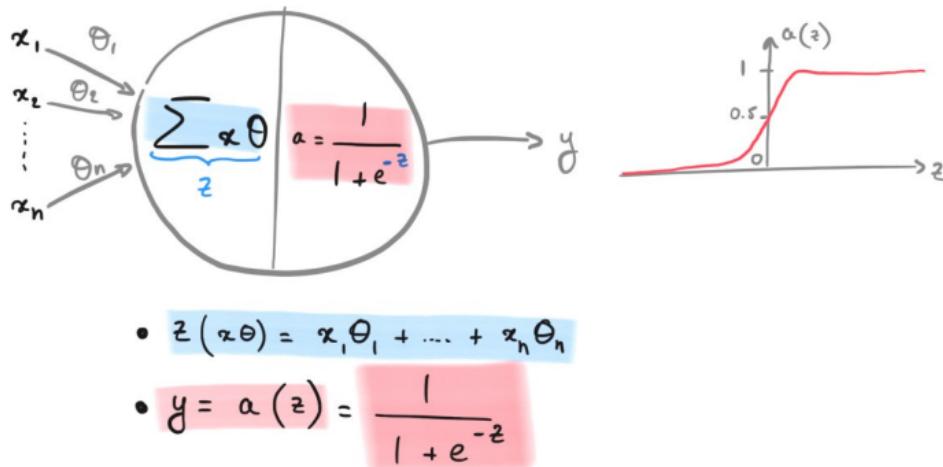
Analysons ce qui se passe dans **1 neurone** :

Le réseau de neurone le plus simple qui existe porte le nom de **perceptron**. Il est **identique** à la **Régression Logistique** vu précédemment.

Les entrées du neurone sont les **features x** multipliées par des **paramètres Theta** à apprendre. Le calcul effectué par le neurone peut être divisé en 2 étapes :

1- Le neurone calcule la **somme z** de toutes les entrées $z = \sum x \Theta$. C'est un calcul linéaire.

2- Le neurone passe z dans sa **fonction d'activation**. Ici la fonction sigmoïde (fonction Logistique). C'est un calcul non-linéaire.



Note :

On utilise souvent d'autres fonctions d'activation que la fonction sigmoïde pour simplifier le calcul du gradient et ainsi obtenir des cycles d'apprentissage plus rapide :

- La fonction tangente hyperbolique « $\tanh(z)$ »
- La fonction « $\text{Relu}(z)$ »

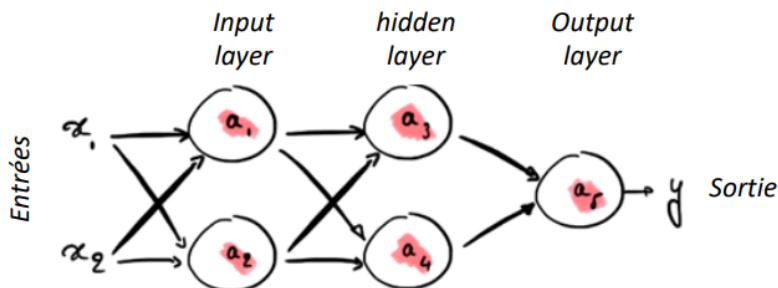
Réseaux à plusieurs Neurones : le Deep Learning

Pour créer un réseau de neurones, il suffit de développer plusieurs de ces perceptrons et de les connecter entre eux d'un façon particulière :

- On réunit les neurones en **colonne** (on dit qu'on les réunit en couche (en **layer**)). Au sein de leur colonne, les neurones ne sont pas connectés entre eux.
- On connecte toutes les **sorties** des neurones d'une colonne à gauche aux **entrées** de tous les neurones de la colonne à droite qui suit.

On peut ainsi construire un réseau avec autant de couches et de neurones que l'on souhaite. Plus il y a de couches, plus on dit que le réseau est profond (**deep**) et plus le modèle devient **riche**, mais aussi difficile à entraîner. C'est ça le Deep Learning.

Voici un exemple d'un réseau à 5 neurones (et **3 layers**). Tous les **layers** entre la couche d'entrée et la couche de sortie sont dits cachés car nous n'avons pas accès à leurs entrées/sorties, qui sont utilisées par les **layers** suivants.



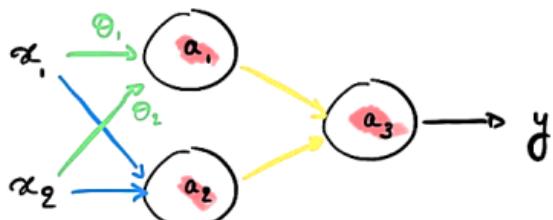
Dans les détails, un réseau plus simple (3 neurones) nous donnerait la sortie suivante :

$$a_3 = \sigma(\theta_1 a_1 + \theta_2 a_2)$$

où Theta1 et Theta2 sont les coefficients liés aux connections entre neurones $a_1 \Rightarrow a_3$ et $a_2 \Rightarrow a_3$.

Ce sont les **paramètres** de notre modèle.

Dans le réseau suivant on a donc 6 paramètres (ici différencié par des couleurs mais la réelle annotation est plus complexe) :



$$a_1 = \sigma(\theta_1 x_1 + \theta_2 x_2)$$

$$a_2 = \sigma(\theta_1 a_1 + \theta_2 a_2)$$

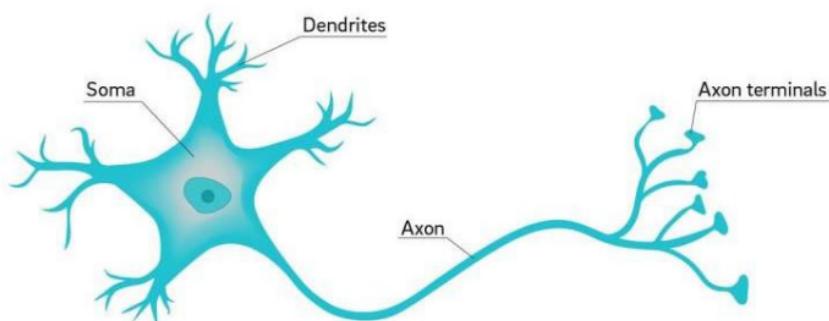
$$a_3 = \sigma(\theta_1 a_1 + \theta_2 a_2)$$

C'est comme le cerveau humain ?

On a longtemps fait le rapprochement entre le cerveau humain et les Neural Network pour démontrer la puissance de ces algorithmes. Voici l'analogie qui est encore présentée aujourd'hui aux novices :

La fonction d'activation produit une sortie si les entrées qu'elle reçoit dépassent un certain seuil, à la manière qu'un neurone biologique produit un signal électrique en fonction des stimulus qu'il reçoit aux **Dendrites** (ce sont les entrées du neurones).

Dans un Neurone, ce signal circule jusqu'aux différents terminaux de **l'axone** pour être transmis à d'autres neurones, tout comme la fonction d'activation envoie sa sortie aux neurones du niveau suivant.



En réalité, les Réseaux de Neurones n'ont rien à voir avec le cerveau humain ! (on utilise juste l'analogie pour expliquer)

Un Réseau de Neurone n'est en fait qu'une énorme **composée** de milliers de fonctions mathématiques, et aujourd'hui les neuroscientifiques ont démontré que le fonctionnement du cerveau dépasse de loin l'architecture « **simpliste** » des réseaux de neurones.

On utilise l'expression réseau de neurone sûrement par facilité car l'analogie avec les neurones facilite la compréhension de ce type

de modèle mais aussi car ça a permis d'attirer l'intérêt des médias, donc une histoire de marketing finalement.

L'entraînement d'un Réseau de Neurone :

Rappel, pour résoudre un problème de Supervised Learning, il faut les 4 éléments suivants :

- 1- Un Dataset
- 2- Un Modèle et ses paramètres
- 3- Une Fonction Coût et son gradient
- 4- Un Algorithme de minimisation (Gradient Descent)

→ Pour le **Dataset** pas de problème, il suffit de disposer d'un tableau (X, y) comme pour les autres problèmes. Les features ($x_1, x_2, x_3 \dots$) sont distribuées à l'entrée du réseau (dans le premier layer)

→ Pour programmer le **Modèle**, il faut emboîter les fonctions des différents niveaux d'activation les unes dans les autres (comme on a vu avec l'exemple des 3 neurones). C'est ce qu'on appelle la **Forward Propagation** (faire le chemin des entrées X vers la sortie y)

→ Pour exprimer la **Fonction Coût** et son gradient, c'est mathématiquement délicat. Il faut calculer la contribution de **chaque neurone** dans **l'erreur finale**. Pour cela, on va utiliser une technique nommée la **Back Propagation** (faire le chemin dans le sens inverse : y vers X).

→ Enfin, pour **minimiser la Fonction Coût**, il suffit d'utiliser le Gradient Descent en utilisant les **gradients** calculés avec la **Back Propagation**. Le Gradient Descent en lui-même n'est pas différent de celui qu'on a vu précédemment.

Programmer son premier Réseau de Neurones (pour identifier des espèces d'Iris) :

Normalement on développe des Réseaux de Neurones avec un framework comme Tensorflow mais ici on va se contenter de Sklearn.

Il faudra donc importer « MLPClassifier » qui signifie (Multi-Layer Perceptron Classifier).

Pour l'exemple ici, l'algorithme va utiliser 4 features pour effectuer son calcul :

- x1 : la longueur du pétale
- x2 : la largeur du pétale
- x3 : la longueur du sépale
- x4 : la largeur du sépale

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.neural_network import MLPClassifier

# charger les données
iris = load_iris()

X = iris.data
y = iris.target

X.shape # notre Dataset comprend 150 exemples et 4 variables

# Visualisation des données
colormap=np.array(['Red','green','blue'])
plt.scatter(X[:,3], X[:,1], c = colormap[y])
```

Pour développer un réseau à 3 hidden layers et 10 neurones dans chaque layer, on utilise ici le code suivant :
« hidden_layer_sizes=(10, 10, 10) »

```
# Création du modèle
model = MLPClassifier(hidden_layer_sizes=(10, 10, 10), max_iter=1000)
model.fit(X, y)
model.score(X, y)
```

Normalement on doit obtenir un score de 98% pour notre Neural Network. Donc la machine arrive à prédire la bonne espèce de fleur 98% du temps.

Résumé de l'apprentissage supervisé :

On utilise beaucoup l'apprentissage supervisé sur les **réseaux de neurones** (la technique la plus utilisée en ML et DeepLearning).

Rappel : Il existe 2 familles de problèmes dans l'apprentissage supervisé :

- Les Régressions
- Les Classifications

Pour résoudre ces problèmes, il y a 4 étapes essentielles :

- 1- Le Dataset
- 2- Le Modèle
- 3- La Fonction Coût
- 4- L'algorithme de minimisation

Avec Sklearn on peut développer des modèles de ML simplement en utilisant les fonctions que nous avons vus et qui intègrent directement les étapes 3 et 4. Il reste plus qu'à :

- 1- Importer le Dataset
- 2- Choisir un modèle parmi ceux proposés par Sklearn
- 3- Utiliser la fonction `.fit()` pour effectuer l'apprentissage

Il existe beaucoup d'algorithmes de Régressions et Classifications que ceux que nous avons vus ! (regardez sur internet si besoin)

6- Apprentissage Non-Supervisé :

(Unsupervised Learning)

Le problème de l'Apprentissage Supervisé

D'un certain point de vue, l'apprentissage supervisé consiste à enseigner à la machine des choses que l'on **sait déjà**, étant donné

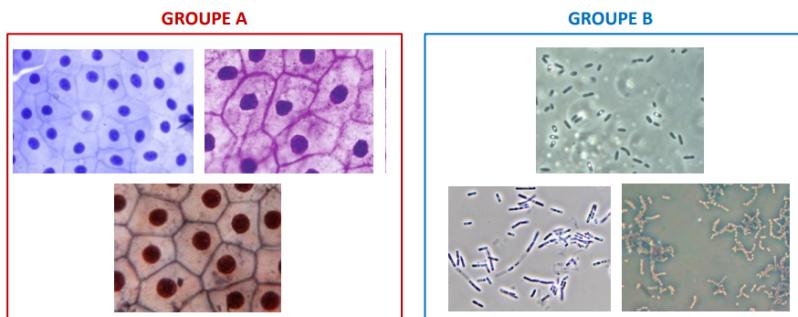
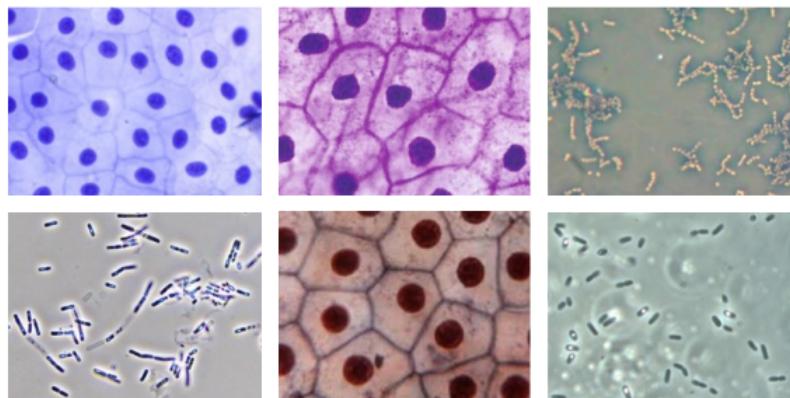
qu'on construit à l'avance un Dataset qui contient des questions X et des réponses y.

Que faire alors si nous n'avons pas de valeur y ?

Que faire si vous voulez que la machine vous aide à compléter vos connaissances en apprenant certaines choses que vous ignorez ?

Comment alors apprendre sans exemple de ce qu'il faut apprendre ?

Regardons ces 6 photos, essayons de les regrouper en 2 familles selon leur ressemblance :



Nul besoin de savoir s'il s'agit de bactéries, cellules animales ou protéines pour classer ses images.

Notre cerveau a reconnu des **structures communes** dans les données et les a regrouper selon juste ce facteur de vision.

C'est la même chose pour la machine en apprentissage non-supervisé.

On va disposer d'un Dataset (x) sans valeur (y), et la machine **apprend** à reconnaître des **structures** dans les données (x) qu'on lui fournit.

On peut ainsi regrouper des données dans des **clusters** (c'est le **Clustering**), détecter des anomalies ou encore réduire la dimension de données très riches en compilant les dimensions ensembles.

Algorithme de K-Mean Clustering :

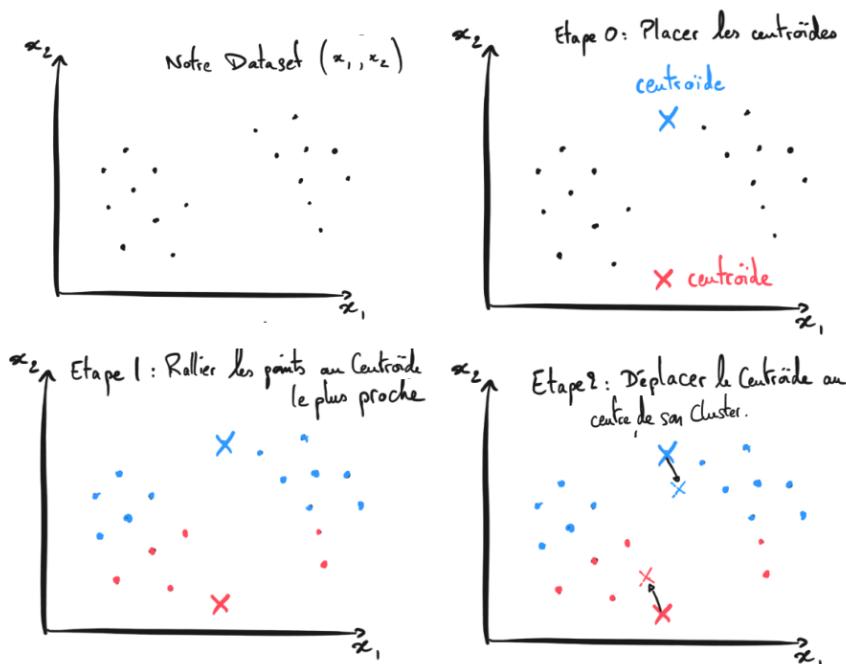
Le K-Mean Clustering est sans doute l'algorithme le plus populaire pour les problèmes de **Clustering** (il regroupe les données selon leur structure commune).

Il est souvent utilisé en marketing pour cibler des groupes de clients semblables pour certaines campagnes publicitaires.

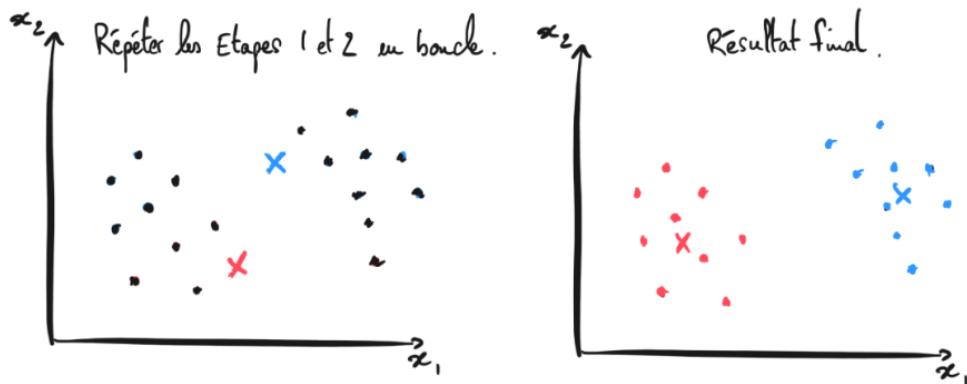
L'algorithme fonctionne en 2 étapes répétées en boucle.

D'abord, on place au hasard un nombre **k** de **centres** dans notre nuage de points. Dans l'exemple suivant $k=2$ et ensuite :

- Étape 1 : on rallie chaque exemple au centre le plus proche.
- Après on a donc nos K-Clusters (ici 2 clusters)
- Étape 2 : On déplace les centres au milieu de leur Cluster.



On répète les étapes 1 et 2 en boucle jusqu'à ce que **les centres ne bougent plus**.



Programmer un K-Mean Clustering :

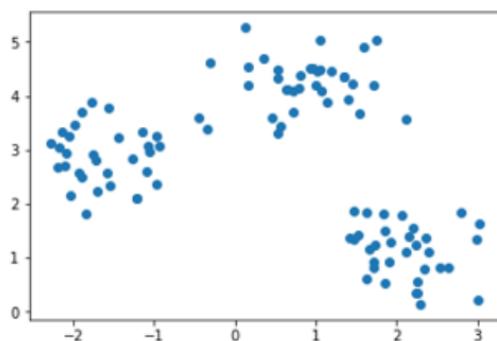
La librairie Sklearn nous permet aussi de faire du Unsupervised Learning.

La fonction « make_blobs » permet de simuler des clusters dans un Dataset.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import KMeans
```

Pour cet exemple, on va créer un Dataset de 100 exemples à 2 features en simulant 3 clusters :

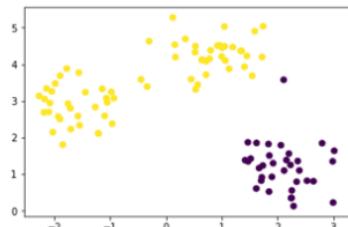
```
# Générer des données:
X, y = make_blobs(n_samples=100, centers = 3, cluster_std=0.5, random_state=0) #nb_features = 2 par défaut
plt.scatter(X[:,0], X[:, 1])
```



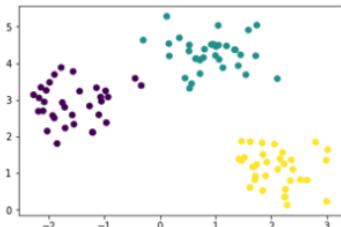
On va ensuite entraîner un modèle de K-Mean Clustering à 3 centres (K=3). On affiche aussi les résultats qu'on obtient pour K=2 et K=4 :

```
# Entrainer le modèle de K-mean Clustering  
model = KMeans(n_clusters=3)  
model.fit(X)
```

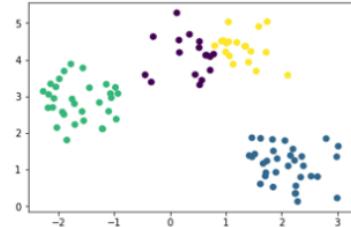
```
#Visualiser les Clusters  
predictions = model.predict(X)  
plt.scatter(X[:,0], X[:,1], c=predictions)
```



K = 2



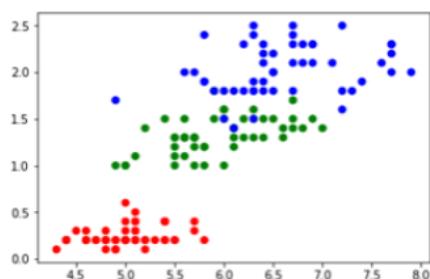
K = 3



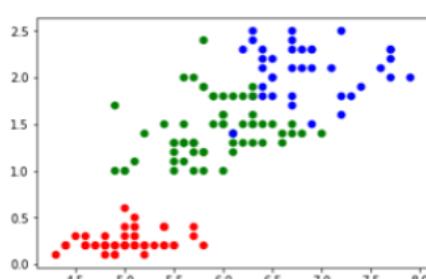
K = 4

Si un jour on tombe sur un tableau de données sur les caractéristiques d'un produit, ou sur des documents qu'on doit classer, utiliser le K-Mean Clustering pour laisser la machine proposer sa méthode de classement peut être une très bonne chose, elle peut proposer une méthode de classement même plus efficace que celle qu'on a imaginé.

Pour finir, prenons le Dataset des fleurs d'Iris qu'on a classer grâce à un réseau de neurones dans le chapitre 5 et voyons les clusters produits par l'algorithme de K-Mean Clustering.



Neural Network



K-Mean Clustering

On voit que les résultats sont très ressemblants, donc voici une preuve qu'il est possible de faire de bonnes classifications même

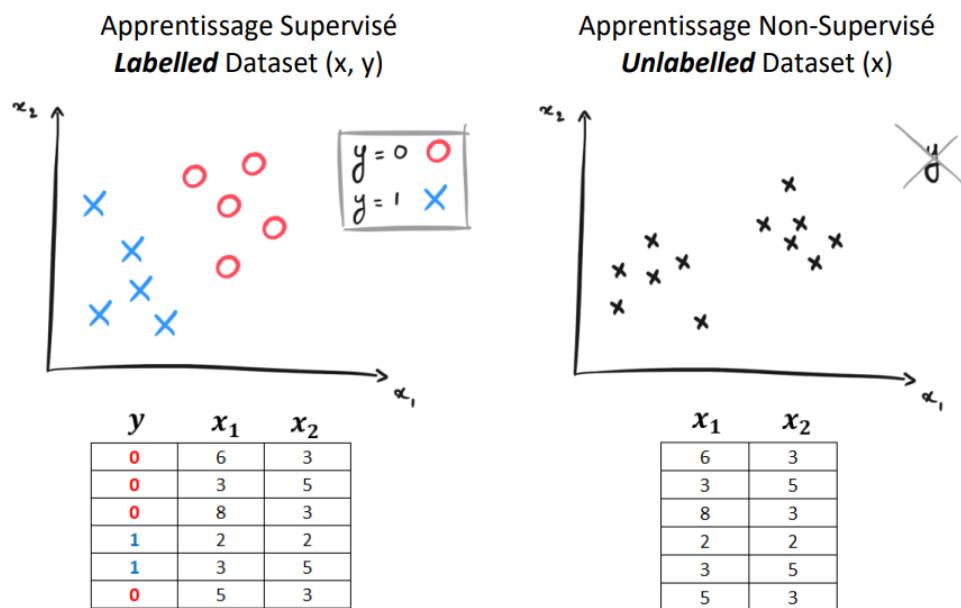
avec des Datasets (x) sans valeur (y) en utilisant l'apprentissage **non-supervisé**.

Conclusion sur le Unsupervised Learning :

Dans l'apprentissage supervisé, la machine reçoit un Dataset où les exemples (x) sont **étiquetés** d'une valeur (y), on appelle ça un **Labelled Dataset**).

Il est alors possible de trouver une relation générale qui relie (x) à (y).

Dans l'apprentissage non-supervisé, nous ne pouvons pas faire cela, parce qu'il manque la variable (y) à notre Dataset. Il est donc **Unlabelled**.



Mais cela ne nous empêche pas de pouvoir **segmenter** le Dataset en différents **Clusters** grâce au K-Mean Clustering ou bien de détecter des **anomalies** en calculant des densités de probabilités. Avec d'autres algorithmes (le **Principal Component Analysis**) on peut aussi réduire efficacement les dimensions d'un Dataset.

Avec ces méthodes, il est possible de segmenter un marché, de développer des systèmes de détection de fraude bancaire, ou d'aider la recherche scientifique.

7- Comment gérer un projet de Machine Learning ?

Nous allons parler des 2 problèmes les plus courants en Machine Learning :

- Une mauvaise préparation du Dataset
- Le problème d'Over fitting

« Le plus important, ce n'est pas l'algorithme, ce sont les **Données** » Andrew Ng

Une étude faite en 2001 par Michelle Banko et Eric Brill montre que :

La performance d'un programme en ML dépend avant tout de la quantité de données que comporte notre Dataset. Ce qui explique l'obsession des grandes firmes du Web (GAFAM) à récolter des quantités colossales de données.

L'étude révèle que beaucoup d'algorithmes de ML sont **similaires** en termes de performance mais que les données (la qualité et quantité) récoltés va jouer un rôle crucial dans les performances finales de ces différents algorithmes et ainsi en créer des meilleurs que d'autres.

Conséquence :

Si vous avez plus de données et de meilleures données que votre concurrent, vous êtes le **vainqueur** même si votre concurrent a un meilleur algorithme !

Mais avoir beaucoup de données suffit pas il faut en avoir des bonnes et surtout **comprendre** ces données.

C'est ce point qui fait défaut à de nombreux Data Scientists.

Pourtant, l'étape de préparation des données (le Data pre-processing) représente le plus grand temps passé sur un projet de ML.

Data pre-processing : Comment préparer son Dataset ?

Lorsqu'on reçoit un Dataset il faut obligatoirement faire des retouches dessus avant de commencer à faire du ML.

Voici une liste d'action par exemple à compléter avant :

- Il est courant qu'un Dataset contienne des **anomalies** qu'il faut traiter, voir des **erreurs**, qu'il faut **supprimer** pour ne pas biaiser l'apprentissage de la machine.
- Il est important aussi de **normaliser** nos données, c-a-d les mettre sur une même échelle pour rendre l'apprentissage de la machine plus rapide et efficace.
- Si on a des valeurs manquantes, il faut être capable de les gérer (supprimer ou assigner une valeur par défaut).
- Si on a des **features catégoriales** (ex : homme/femme) il faut les convertir en données numériques (homme=0/femme=1)
- Aussi, il faut nettoyer notre Dataset de **features redondantes** (qui ont une forte corrélation) pour faciliter l'apprentissage de la machine.
- Enfin, un point qui peut faire toute la différence, c'est la **création de nouvelles features**, ce qu'on appelle le **feature engineering**.

Exemple : On prend un Dataset immobilier qui contient les features :

- x_1 = longueur jardin
- x_2 = largeur jardin

Alors il est possible de créer $x_3 = x_1 * x_2$ qui équivaut à la surface du jardin.

Typiquement, Sklearn et Pandas disposent des fonctions nécessaires pour faire un bon data pre-processing.

Pour charger un fichier Excel au format CSV et utiliser après Pandas.

```
import pandas as pd

Dataset = pd.read_csv('dataset.csv')
print(Dataset.head()) # afficher le Dataset
```

Précision, il faut utiliser un fichier bien organisé de base :

Exemple à suivre

	A	B	C	D	E	F	G
1	prix	surface	qualite	annee	jardin	garage	distance parc
2	313000	124	3	1.5	0 oui	150	
3	2384000	339	5	2.5	70 oui	1300	
4	342000	179	3	2	0 oui	30	
5	420000	186	3	2.25	30 oui	2175	
6	550000	180	4	2.5	25 oui	400	
7	490000	82	2	1	10 non	380	
8	335000	125	2	2	0 non	160	
9	482000	252	4	2.5	0 oui	210	
10	452500	226	3	2.5	32 oui	230	
11	640000	141	4	2	0 non	25	
12	463000	159	3	1.75	0 oui	60	
13	1400000	271	4	2.5	80 oui	80	
14	588500	216	3	1.75	0 oui	55	
15	365000	101	3	1	0 non	96	
16	1200000	270	5	2.75	50 oui	145	
17	242500	111	3	1.5	5 non	1450	

VS

Exemple à NE PAS suivre

	A	B	C	D	E	F	G	H	I
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									
12									
13									
14									
15									
16									
17									
18									
19									
20									
21									
22									
23									
24									

On peut ensuite utiliser certaines fonctions pour **nettoyer** notre Dataset, convertir les **catégories** en valeurs numériques, et charger notre Dataset dans une matrice X et un vecteur y pour commencer le Machine Learning !

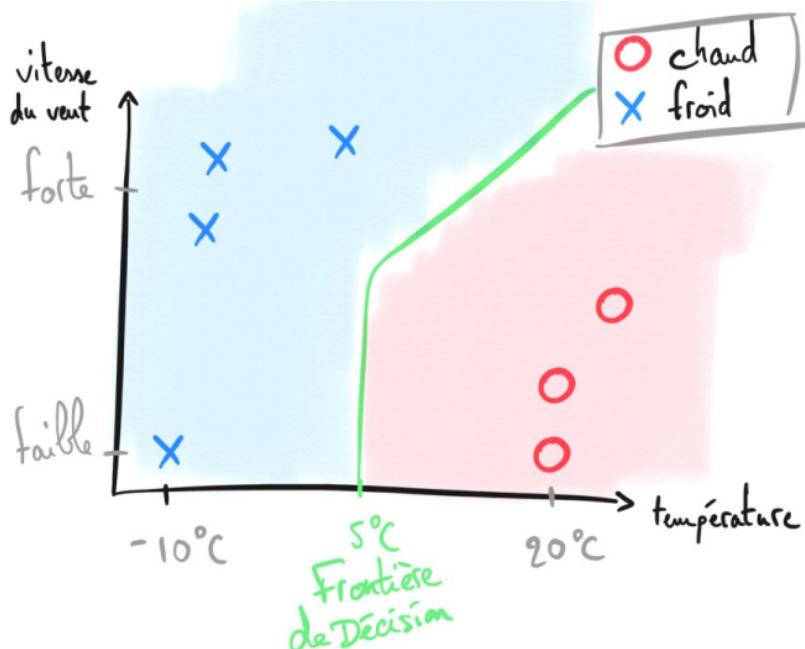
```
Dataset = Dataset.fillna(value=99999) # assigner une valeur défaut
Dataset = pd.get_dummies(Dataset) # remplacer les catégories

y = Dataset['prix'].values # Créer le vecteur target y
X = Dataset.drop(['prix'], axis=1).values # Créer la matrice features X
```

L'expertise est cruciale

Jouer ainsi avec les données peut s'avérer dangereux si le Data Scientist n'a pas de connaissances techniques sur l'application finale : finance, médecine, ingénierie, climatologie etc

Exemple : En donnant le Dataset suivant à un modèle de ML, vous obtiendrez une frontière de décision qui indique qu'il fait chaud quand la température est supérieure à 5°C...



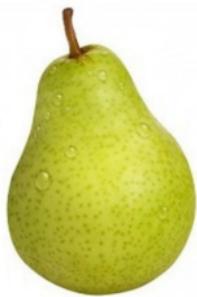
Dans cette application aussi simple, on a une certaine expertise qui permet de rejeter la réponse de la machine, et on comprend que le Dataset a besoin d'être complété avec des données supplémentaires.

Conclusion : Pour être le meilleur Data Scientist il faut se rattacher sur des projets dont on a les **compétences techniques** et la **réflexions** ainsi que les **connaissances** sur le sujet.

Les données doivent toujours venir de la même distribution

Une dernière erreur qu'on va voir ici qui peut causer une chute de performance assez importante est l'utilisation d'un modèle de ML sur des données provenant d'une **autre distribution** que les données sur lesquelles la machine a été entraînée.

Exemple : si on développe un modèle pour reconnaître des poires en donnant à la machine des photos Haute Définition de poires bien droites et sans taches, mais que la machine utilise ensuite une mauvaise caméra qui déforme les couleurs et qui voit les poires toutes empilées les unes sur les autres, le modèle ne pourra pas reconnaître les features qu'il a appris durant l'entraînement avec la même précision.



Ce que vous aviez donné à apprendre



Ce que la machine voit après l'apprentissage

Conclusion :

Il est important de bien **préparer** son Dataset, en **supprimant les défauts** qu'il contient, en s'assurant qu'il représente des données provenant de la **même distribution** que pour l'application finale, et en **comprenant** en profondeur le **sens des données** dont on dispose.

Le travail de préparation des données prend souvent **80 % du temps de travail** d'un Data Scientist, mais s'il est bien fait, vous n'aurez aucun problème par la suite.

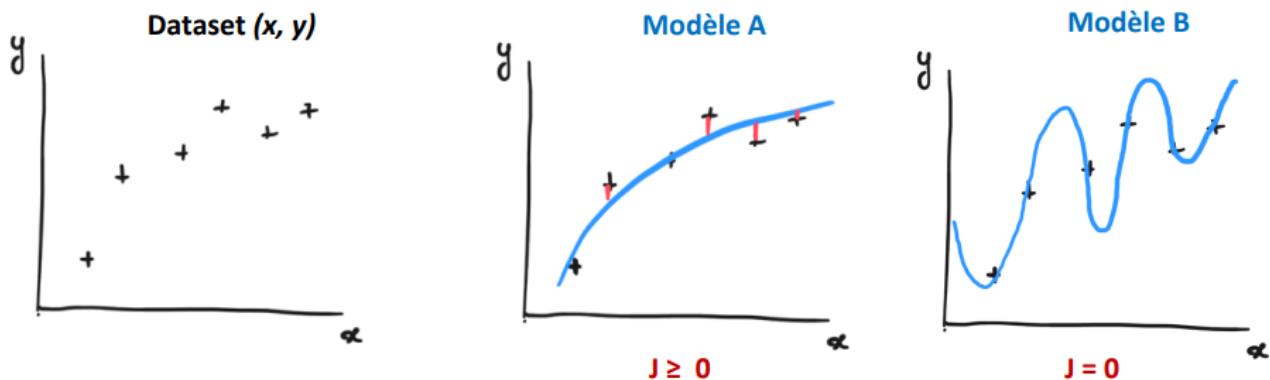
Plus aucun problème sauf 1 : **l'Over fitting !**

Over fitting et Régularisation

Dans le chapitre sur l'apprentissage supervisée on a développé des modèles en cherchant à **minimiser** les erreurs avec le Dataset.

Mais ce n'est pas si simple en pratique..

Exemple :



Ici on voit que le **modèle B** ne donne **aucune erreur** par rapport au Dataset, donc d'après ce que nous avons vu il **devrait être parfait** !

Pourtant c'est le **modèle A** qui semble plus **convaincant**, alors même que celui-ci donne une **Fonction Coût plus élevée**.

Le modèle B souffre ici **d'Over fitting** (surentraînement), qui est un phénomène **très courant** en ML et qu'il **faut absolument éviter**.

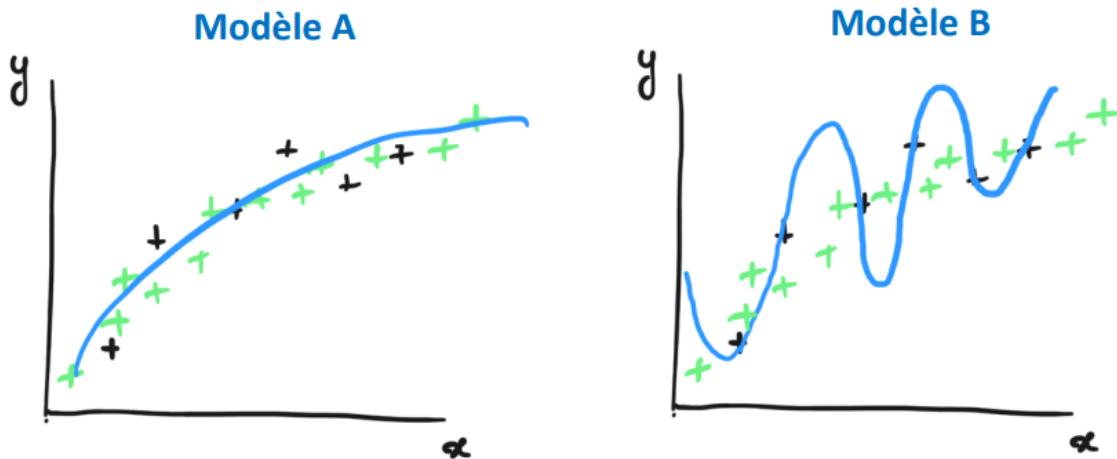
Over Fitting

On parle d'Over fitting pour dire que le **modèle s'est trop spécialisé sur les données** qui lui ont été fournies et a **perdu tout sens de généralisation**.

Un Over fitting survient le plus souvent quand un modèle **trop complexe** (avec trop de paramètres ou trop de features) a été entraîné.

Dans ce cas, le modèle a certes un faible coût $J(\Theta)$, mais il a aussi ce qu'on appelle une **grande variance**.

Conséquence : un modèle moins performant que prévu quand on le soumet à de nouvelles données et une machine qui confond les pommes et les poires par exemple.

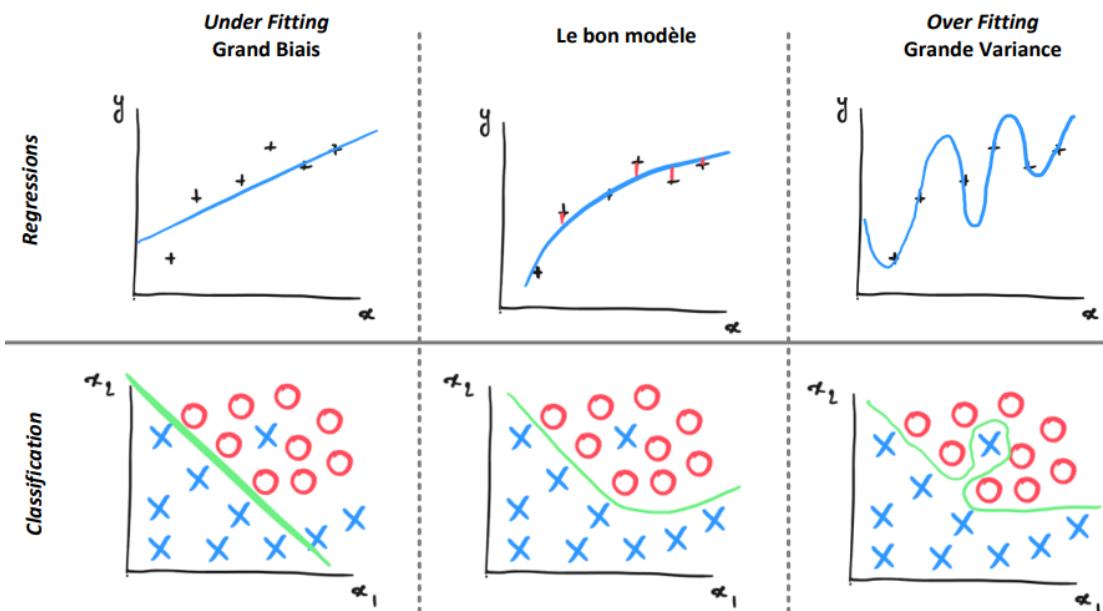


Sur le Dataset B était meilleur que A.
Mais sur de nouvelles données, A est meilleur que B

On pourrait alors se dire qu'il suffit de développer des modèles moins complexes avec moins de features et on aurait plus de problème de variance !

C'est vrai, mais on risque d'avoir alors un modèle erroné qui manque de précision. On appelle ça **Under fitting** (sous-entraînement) et on dit que le modèle a un **grand biais**.

Ce problème touche à la fois les régressions et les classifications :



Comment trouver alors le **juste milieu entre biais et variance** ?
C'est une des grandes questions à laquelle sont confrontés les Data Scientists.

Il existe une méthode qui permet de garder toutes les features d'un modèle tout en **régulant l'amplitude** des paramètres Theta.
Cette méthode s'appelle la **Régularisation**.

La Régularisation

Elle permet de limiter la variance d'un modèle sans sacrifier son biais. Pour cela, différentes techniques existent :

1-On peut légèrement **pénaliser la Fonction Coût** du modèle en **ajoutant** un **terme de pénalité** sur ses paramètres.

Pour la régression linéaire, la Fonction Coût devient alors :

$$J(\theta) = \frac{1}{2m} \sum (F(X) - Y)^2 + \lambda \sum \theta^2 \quad (\text{Ridge ou L2 Régularisation})$$

Le facteur de régularisation **lambda** correspond au niveau de pénalité :

S'il est trop grand, on risque l'Under fitting.

S'il est trop faible, c'est l'Over fitting.

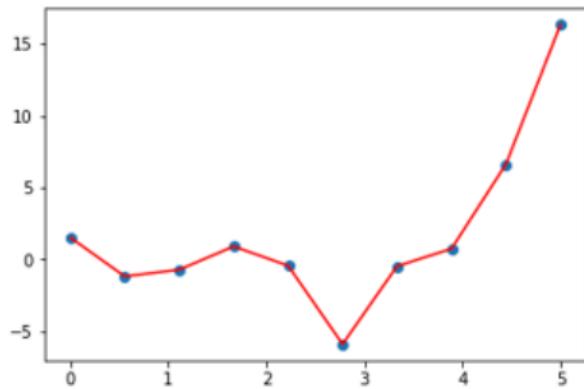
On peut le contrôler directement dans Sklearn.

2- Pour le K-Nearest Neighbour, on peut **augmenter la valeur de K** (nombre de voisins). Le modèle ne tient alors pas compte des anomalies noyées dans la masse.

3- Pour les Réseaux de Neurones, une technique nommée **Dropout** pénalise le modèle en désactivant aléatoirement certains neurones à chaque cycle de Gradient Descent. Le réseau perd alors légèrement de ses facultés et est moins sensible aux détails.

Voici ce que la Régression linéaire classique nous donnera en développant un modèle polynomial de degré 10 sur le Dataset suivant :

```
Coeff R2 = 1.0
[<matplotlib.lines.Line2D at 0x2300c670ef0>]
```



```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Creation d'un Dataset x, y
np.random.seed(0)
x = np.linspace(0, 5, 10)
y = x - 2 * (x ** 2) + 0.5 * (x ** 3) + np.random.normal(-2, 2, 10)
plt.scatter(x, y)

# Creation de plusieurs features pour notre modele
X = x[:, np.newaxis]
X = PolynomialFeatures(degree=10, include_bias=False).fit_transform(X)
X.shape

# Entrainement du modele. Ici on utilise les Equations Normales (LinearRegression)
# Les Equations normales reposent sur la méthode des moindres carrés, c'est plus
# rapide que le Gradient Descent.
model = LinearRegression()
model.fit(X,y)
print('Coeff R2 =', model.score(X, y))
plt.scatter(x, y, marker='o')
plt.plot(x, model.predict(X), c='red')
```

Dans Sklearn, on peut développer un modèle avec régularisation grâce au modèle **Ridge** : « Sklearn.linear_model.Ridge »
Le modèle a certes un coefficient R^2 plus faible, mais il produit une meilleure généralisation et fera donc moins d'erreurs sur les données futures.

```

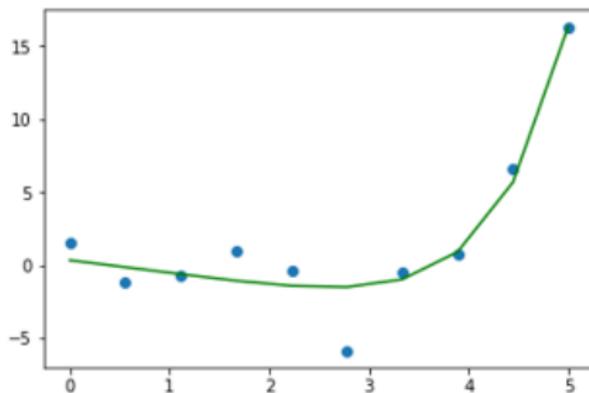
from sklearn.linear_model import Ridge

ridge = Ridge(alpha=0.1) # alpha est le facteur de régularisation.
ridge.fit(X,y)
print('Coeff R2 =', ridge.score(X, y))

plt.scatter(x, y, marker='o')
plt.plot(x, ridge.predict(X), c = 'green')

```

Coeff R2 = 0.9132413559476188
[<matplotlib.lines.Line2D at 0x2300c89c0f0>]



Maintenant on sait que la **performance réelle** d'un modèle de ML ne repose pas simplement sur sa Fonction Coût : ***on risque l'Over fitting*** chaque fois qu'un modèle se **spécialise** trop sur les données qu'on lui **donne** à étudier.

Mais comment être sûr de la performance que notre modèle aura sur des données futures, c-a-d des données sur lesquelles il n'aura **pas été entraîné** ?

La réponse est dans la question, il faut entraîner son modèle **sur une partie seulement** du Dataset et utiliser la seconde partie pour évaluer la vraie performance de notre modèle.

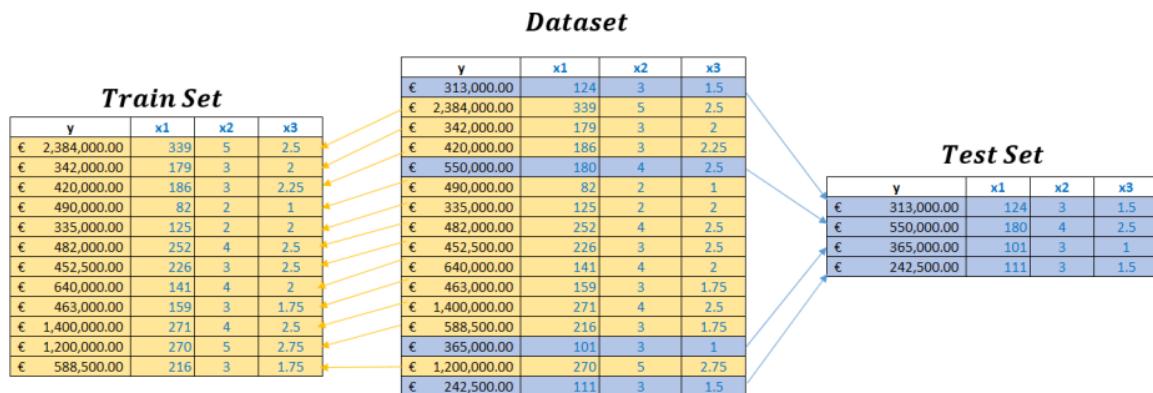
On appelle ça le **Train set** et le **Test set**.

Train set et Test set

La bonne manière de mesurer la performance de son modèle de ML est de tester celui-ci sur des données qui n'ont pas servi à l'entraînement. On divise ainsi le Dataset **aléatoirement** en 2 parties avec un rapport 80/20 :

- Train set (80%), qui permet à la machine d'entraîner un modèle sur la majorité des données.

- Test set (20%), qui permet d'évaluer la performance du modèle sur de nouvelles données.



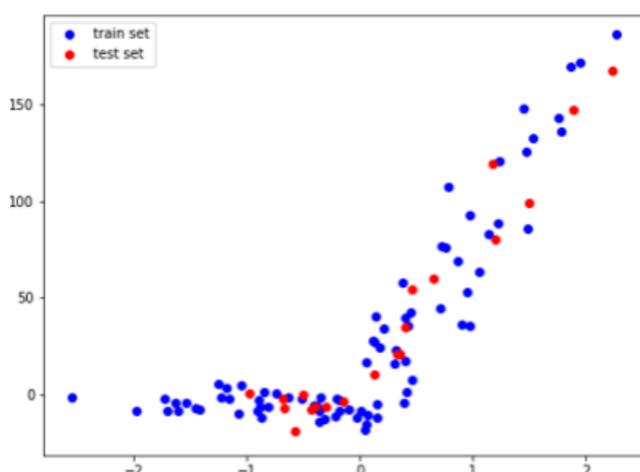
Pour créer un Train et Test set à partir de notre Dataset, on utilise la fonction « train_test_split » de Sklearn :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split

# Creation d'un Dataset Aleatoire
np.random.seed(0)
x, y = make_regression(n_samples=100, n_features=1, noise=10)
y = np.abs(y) + y + np.random.normal(-5, 5, 100)
plt.scatter(x, y)

# Creation des Train set et Test set a partir du Dataset
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

# Visualisation des Train set et Test set
plt.scatter(x_train, y_train, c='blue', label='Train set')
plt.scatter(x_test, y_test, c='red', label='Test set')
plt.legend()
```



On peut ensuite entraîner notre modèle sur le Train Set (Xtrain, Ytrain), puis l'évaluer sur le Test Set (Xtest, Ytest).

Dans le code ci-dessous mettons en pratique tout ce que nous avons vu, on va créer ici un features polynomiales (degré 10) et utiliser la régularisation de Ridge (pour éviter l'Over fitting).

```
X = PolynomialFeatures(degree = 10, include_bias=False).fit_transform(x)
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

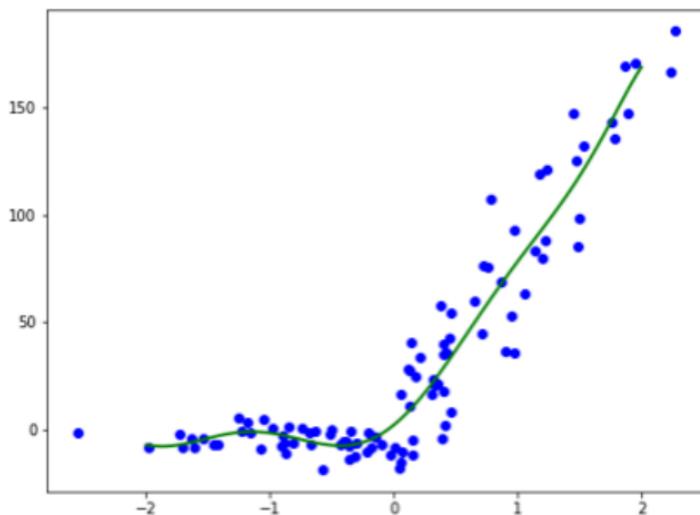
from sklearn.linear_model import Ridge

model = Ridge(alpha = 0.1, random_state=0)
model.fit(x_train, y_train)

print('Coefficient R2 sur Train set:', model.score(x_train, y_train))
print('Coefficient R2 sur Test set:', model.score(x_test, y_test))

plt.figure(figsize=(8,6))
plt.scatter(x, y, c='blue')
a = np.linspace(-2, 2, 100).reshape((100, 1))
A = PolynomialFeatures(degree = 10, include_bias=False).fit_transform(a)
plt.plot(a, model.predict(A), c = 'green', lw=2)

Coefficient R2 sur Train set: 0.9235739839329026
Coefficient R2 sur Test set: 0.9142326950179189
[<matplotlib.lines.Line2D at 0x230151d3080>]
```

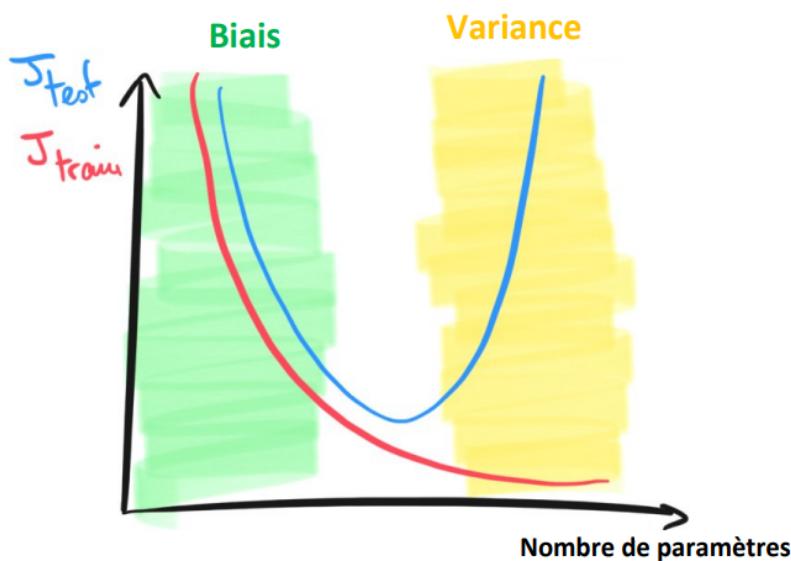


On obtient un score de 92 % pour l'entraînement et un score de 91 % pour l'évaluation sur des données nouvelles.

Repérer un problème d'Under fitting ou d'Over fitting

La technique la plus efficace pour repérer si votre modèle a un **problème de biais (Under fitting)** ou **de variance (Over fitting)** consiste à analyser les **erreurs** (la Fonction Coût) sur le Train set $J(\Theta)\text{train}$ et le Test set $J(\Theta)\text{test}$:

- Si les **erreurs sont grandes** sur le Train set et le Test set, alors le modèle a un grand **biais**, et il faut développer un modèle plus complexe ou développer plus de features.
- Si les erreurs sont faibles sur le Train set mais sont **grandes** sur le Test set, alors le modèle a une grande **variance**.



Que faire en cas d'Over fitting ou d'Under fitting ?

Dans le cas où notre modèle a un grand biais (Under fitting) on peut :

- Créer un modèle plus complexe, avec plus de paramètres
- Créer plus de features à partir des features existantes
- Entraîner notre modèle plus longtemps
- Diminuer le Learning Rate du Gradient Descent (*Rappel* : si le Learning Rate est trop grand, la Fonction Coût ne converge pas)

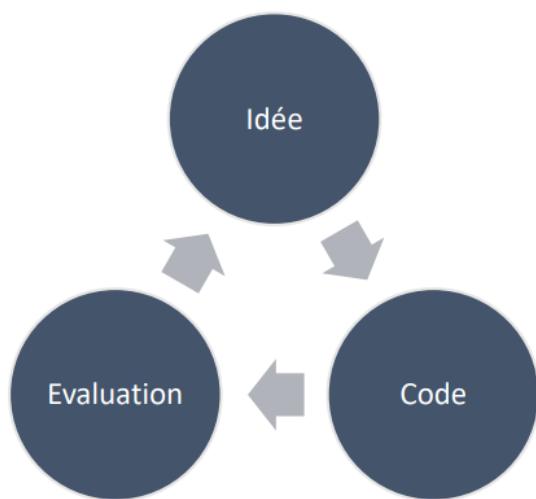
- Récolter plus de features dans les données (parfois une feature importante n'a pas été récoltée)

A l'inverse, si notre modèle a une grande variance (Over fitting) on peut :

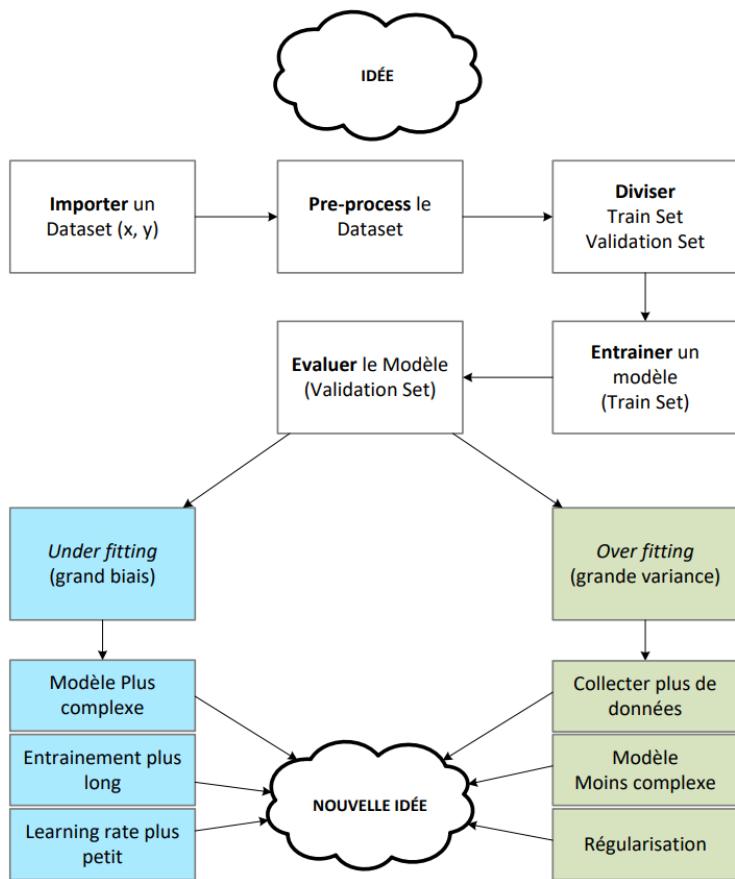
- Utiliser la régularisation
- Utiliser un modèle avec moins de paramètres ou un Dataset avec moins de features
- Collecter plus de données (avoir un Dataset plus grand permet de développer un modèle qui généralise mieux)

Cycle de développement du Machine Learning :

Développer un modèle de ML ne se fait pas du premier coup. On commence souvent avec une première idée de **modèle simple et rapide** à développer, puis on analyse si on a une variance ou un biais et on tente une **nouvelle idée pour corriger** les problèmes rencontrés etc



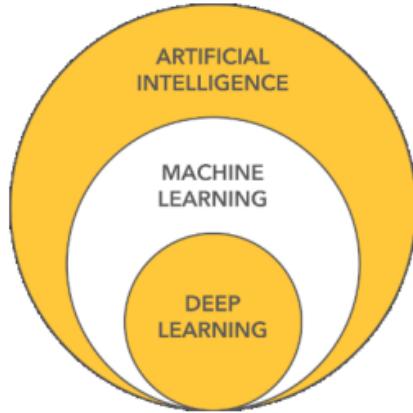
Résumé des étapes de développement en Machine Learning :



Lexique : Formule Résumé du Machine Learning

Différences entre Data Science, Machine Learning et Deep Learning

L'**intelligence Artificielle** est l'ensemble des techniques et théories qui cherchent à développer des modèles capables de **simuler** le comportement humain.



Parmi ces techniques, on trouve le ML très populaire depuis 2010. Le DL (Deep Learning) est un domaine du ML qui est focalisé sur le développement des réseaux de neurones et qui fait face à d'autres défis que ceux du ML.

Parmi ces défis, comment entraîner des modèles avec des millions de paramètres et des milliards de données dans des temps raisonnables.

On dit souvent que ce sont des disciplines de Data Science parce qu'elles utilisent des données pour construire les modèles.

Mais en Data Science, on analyse plus souvent des données pour en créer un modèle en réaction à ces données, alors qu'en ML on crée un programme qui acquiert une aptitude : conduire une voiture, voir des objets etc

Dataset :

Tableau de données (X, y) qui contient 2 types de variables :

- Target y
- Features X

On note m le nombre d'exemples que contient le tableau (le nombre de lignes) et n le nombre de features (le nombre de colonnes X)

Ainsi :

- X est une matrice à m lignes et n colonnes. $X \in \mathbb{R}^{m \times n}$
- y est un vecteur à m lignes. $y \in \mathbb{R}^m$

Pour désigner la feature j de l'exemple i on écrit : $x_j^{(i)}$

Modèle :

Fonction mathématique qui associe X à y , telle que $f(X) = y$.
Un bon modèle doit être une bonne **généralisation**, c-a-d qu'il doit fournir de **petites erreurs** entre $f(x)$ et y sans être sujet à **l'Over fitting**.

On note Theta θ

le **vecteur** qui contient les **paramètres** de notre modèle.

Pour une régression linéaire, la formulation matricielle de notre modèle devient : $F(X) = X.\Theta$

Fonction Coût :

La Fonction Coût **J(Theta)** mesure l'ensemble des erreurs entre le modèle et le Dataset. De nombreuses **métriques d'évaluations** peuvent être utilisés pour la Fonction Coût :

- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)

- Root Mean Squared Error (RMSE)
- Accuracy : pour les classifications
- Precision
- Recall
- F1 score

Rappel : Gradient de la Fonction Coût = $\frac{\partial J(\theta)}{\partial \theta}$

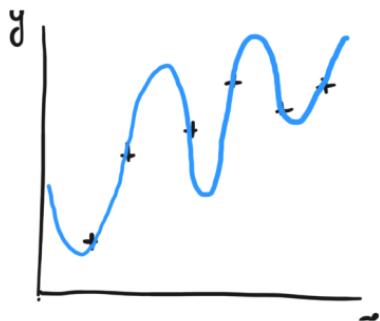
Gradient Descent :

Algorithme de minimisation de la Fonction Coût. Il existe beaucoup de variante de cet algorithme :

- Mini Batch Gradient Descent
- Stochastic Gradient Descent
- Momentum
- RMSProp
- Adam
- etc

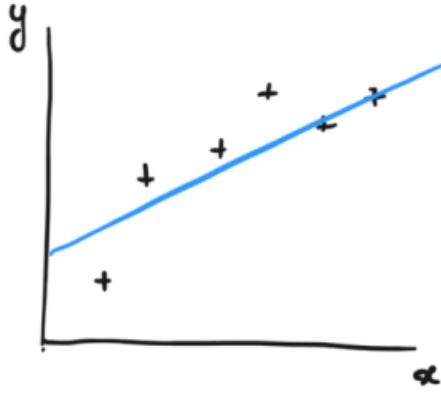
Variance :

C'est l'erreur due à un modèle trop sensible aux détails et incapable de généraliser, ce qui donne lieu à **l'Over fitting**.



Biais :

C'est l'erreur due à un modèle erroné qui manque de précision et donne lieu à un **Under fitting**.



Résumé de la Régression Linéaire :

Dataset: (X, y) avec $X, y \in \mathbb{R}^{m \times n}$

Modèle: $F(X) = X \cdot \theta$

Fonction Coût: $J(\theta) = \frac{1}{2m} \sum (F(X) - y)^2$

Gradient: $\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} X^T \cdot (F(X) - y)$

Gradient Descent: $\theta = \theta - \alpha \times \frac{\partial J(\theta)}{\partial \theta}$

Résumé de la Régression Logistique :

Modèle: $\sigma(X \cdot \theta) = \frac{1}{1 + e^{-X \cdot \theta}}$

Fonction Coût: $J(\theta) = -\frac{1}{m} \sum y \times \log(\sigma(X \cdot \theta)) + (1 - y) \times \log(1 - \sigma(X \cdot \theta))$

Gradient: $\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} X^T \cdot (\sigma(X \cdot \theta) - y)$

Gradient Descent: $\theta = \theta - \alpha \times \frac{\partial J(\theta)}{\partial \theta}$

Code pour visualiser les courbes d'apprentissage sur une régression linéaire :

Importer les libraires :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression
```

Générer un Dataset aléatoire

```
np.random.seed(4)
n = 1
m = 100

x, y = make_regression(n_samples=m, n_features=n, noise=10)
y = y + 100
plt.scatter(x, y)
y = y.reshape(y.shape[0], 1)

#ajouter le Bias a X
X = np.hstack((np.ones(x.shape), x))
X.shape
```

Définir sous forme matricielle le modèle, la Fonction Coût et le gradient.
On définit θ le vecteur qui contient les paramètres a et b .

$$F = X \cdot \theta$$

$$J(\theta) = \frac{1}{2m} \sum (X \cdot \theta - y)^2$$

$$\text{Grad}(\theta) = \frac{1}{m} X^T \cdot (X \cdot \theta - y)$$

```
#definir la fonction modele
def model(X, theta):
    # x shape: (m, n)
    # theta shape: (n,1)
    return X.dot(theta) #shape: (m, 1)

#define la fonction cout
def cost_function(X, y, theta):
    m = len(y)
    J = 1/(2*m) * np.sum((model(X, theta) - y)**2)
    return J

#define la fonction gradient
def gradient(X, y, theta):
    return 1/m * X.T.dot((X.dot(theta) - y))
```

On définit la fonction Gradient Descent avec une boucle `for` :

For all itérations :

$$\theta = \theta - \alpha \cdot \text{Grad}(\theta)$$

```
#algorithme de Gradient Descent
def gradient_descent(X, y, theta, learning_rate = 0.001, iterations = 1000):
    m = len(y)
    cost_history = np.zeros(iterations)
    theta_history = np.zeros((iterations, 2))

    for i in range(0, iterations):
        prediction = model(X, theta)
        theta = theta - learning_rate * gradient(X, y, theta)
        cost_history[i] = cost_function(X, y, theta)
        theta_history[i,:] = theta.T

    return theta, cost_history, theta_history
```

On passe à l'entraînement du modèle, puis on visualise les résultats.

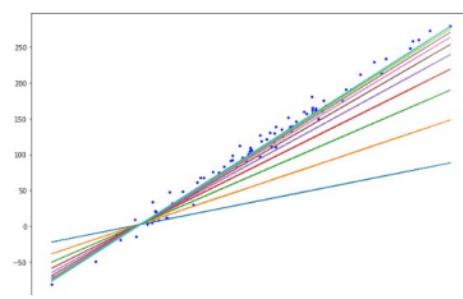
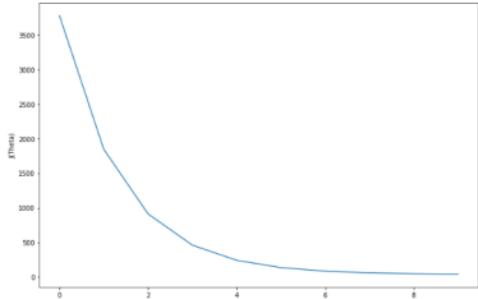
```
# utilisation de l'algorithme
np.random.seed(0)
theta = np.random.randn(2, 1)

iterations = 10
learning_rate = 0.3
theta, cost_history, theta_history = gradient_descent(X, y, theta, learning_rate=learning_rate, iterations = iterations)

#visualisation des courbes d'apprentissage
fig,ax = plt.subplots(figsize=(12,8))

ax.set_ylabel('J(Theta)')
ax.set_xlabel('Iterations')
_=ax.plot(range(iterations),cost_history)

# visualisation du modèle au cours de son apprentissage
fig,ax = plt.subplots(figsize=(12,8))
_=ax.plot(x, y, 'b.')
for i in range(iterations):
   _=ax.plot(x, model(X, theta_history[i]), lw=1)
```



Autres Algorithmes de ML :

Voici d'autres algorithmes très populaires en ML :

- **Support Vector Machine** : Consiste à trouver la frontière de décision linéaire qui éloigne le plus les classes l'une de l'autre. Il est facile de créer des modèles aux dimensions infinies avec cette méthode.
- **Decision Tree** : Consiste à ordonner une série de tests dans un arbre pour arriver à une conclusion.
- **Random Forest** : Un ensemble de Decision Tree construits aléatoirement (avec Bootstrap) qui chacun émet sa solution au problème. La solution majoritairement choisie par la forêt l'emporte.
- **Naive Bayes** : Repose sur l'inférence de Bayes (probabilités conditionnelles)
- **Anomaly Detection Systems** : Algorithme de Unsupervised Learning qui consiste à détecter des anomalies grâce aux densités de probabilités
- **Principal Component Analysis** : Technique de réduction de dimension qui consiste à réduire le nombre de variables de notre Dataset en créant des variables non-correlées.

Conseil pour faire du Machine Learning :

Conseil #1 :

Noter toujours les dimensions de son problème.

100 % des Data Scientists ont déjà fait cette erreur, **ne pas noter sur une feuille les dimensions des matrices du problème.**

A un moment ou un autre, on aura un bug parce qu'on tente de multiplier 2 matrices A et B aux dimensions incompatibles.
Il est alors très utile de vérifier les dimensions des matrices du Dataset, Train set, Test set et des paramètres en utilisant la fonction « numpy.shape » pour les comparer à nos calculs.

```
In [448]: np.random.seed(0)
x, y = make_regression(n_samples=100, n_features=2, noise=10)
x.shape
```

Out[448]: (100, 2)

Conseil #2 :

Loi de Pareto

80 % des effets sont le produit de 20 % des causes.

En ML, souvent 80 % des erreurs de notre modèle sont produites par 20 % des prédictions, ou bien 20 % des points du Dataset, ou bien 20 % des hyper-paramètres etc

De la même manière, 80 % de la performance de notre programme est atteinte après seulement 20 % de travail.

Il faut faire le diagnostic de notre système en cherchant les points 80/20.

Si on veut améliorer de 80 % en produisant seulement 20 % d'efforts, bon courage !

Conseil #3 :

Philosophez et ne pas être obsédés par la performance.

En ML, il est important de savoir prendre du recul. Selon l'application, une précision de 98 % est aussi bien qu'une précision de 98,1 %.

Pourtant, beaucoup de Data Scientists peuvent perdre des semaines à gagner ce petit 0,1 %... pour pas grand-chose. Garder l'esprit ouvert et passer plus de temps à chercher des solutions à d'autres problèmes plus importants.