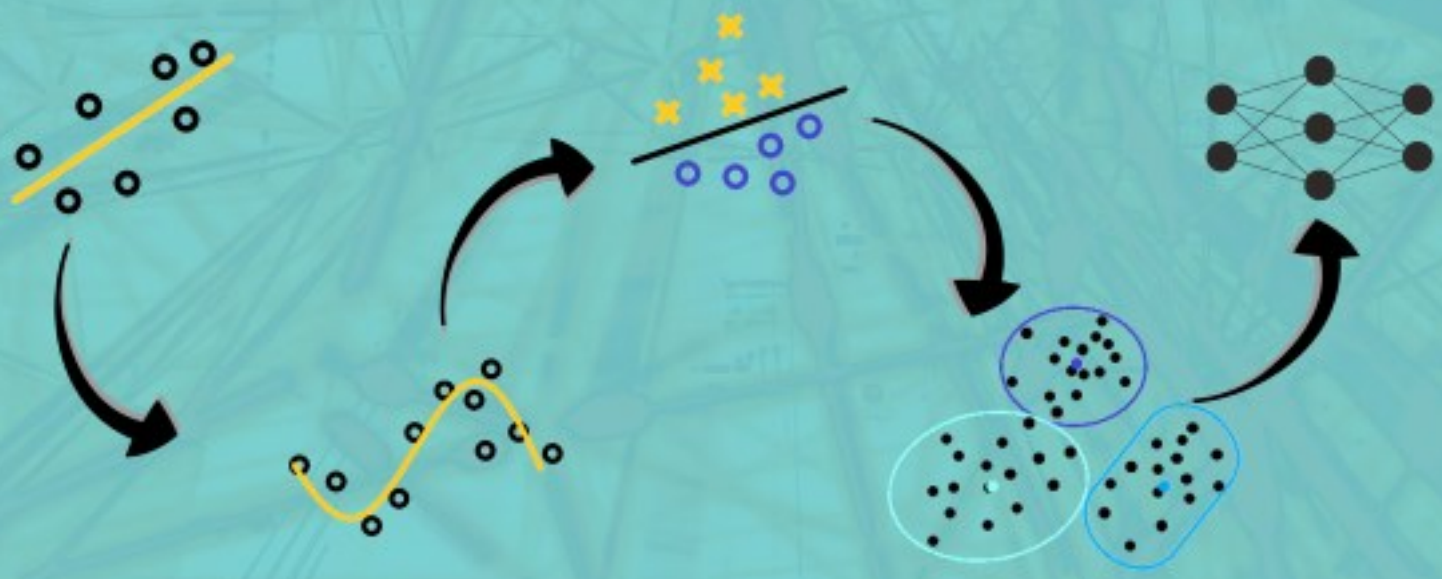
APPRENDRE LE MACHINE LEARNING EN UNE SEMAINE



Guillaume Saint-Cirgue

Table des matières

Introduction – Pourquoi devez-vous lire ce livre ?

En 2019, Le **Machine Learning** est tout autour de nous. Il intervient chaque fois que nous cherchons un mot dans Google, une série sur Netflix, une vidéo sur YouTube, un produit sur Amazon.

Grâce au Machine Learning, des millions de cancers peuvent être diagnostiqués chaque année, des milliards de spams et de virus informatiques sont bloqués pour protéger nos ordinateurs, et sans lui la voiture autonome n'existerait peut-être jamais.

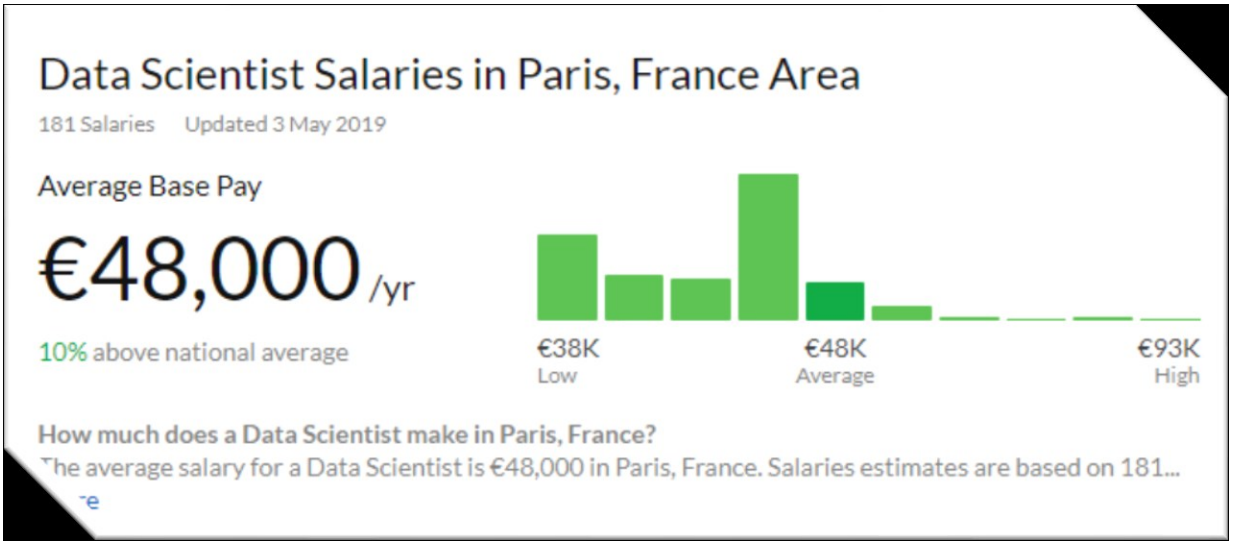
Pourtant le grand public, qui lui donne à tort le nom « **Intelligence Artificielle** », en ignore presque tout. Et il est bien connu que l'Homme a peur de ce qu'il ne comprend pas.

En lisant ce guide, je vous invite à un voyage qui va vous permettre de briser la glace avec l'Intelligence Artificielle et d'apprendre réellement une nouvelle **compétence professionnelle :** Le **Machine Learning**.

Après avoir lu ce guide, vous ferez partie des pionniers d'un nouveau monde, vous donnant accès à des opportunités professionnelles **extraordinaires**, et vous aurez développé votre capacité à **résoudre des problèmes**.

Quel que soit votre métier (Ingénierie, Marketing, Finance, ou même artiste) ce livre vous sera utile, j'en suis convaincu.

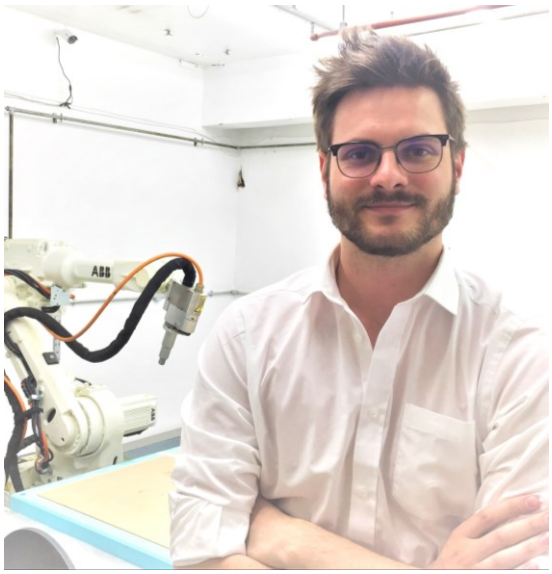
Harvard Business Review, 2012



Salaire de **base moyen** pour un Data Scientist à Paris. Glassdoor, 2019.

Qui suis-je ?

Je m'appelle Guillaume Saint-Cirgue et je suis ingénieur en Machine Learning au Royaume-Uni.



J'exerce ce métier alors que, **comme vous** peut-être, je n'ai pas eu la chance de recevoir des cours d'Intelligence Artificielle au lycée, ni même dans les études supérieures.

J'ai dû tout apprendre de moi-même, en investissant mon temps et mon argent dans des formations du MIT et de Stanford et en passant des week-end entiers à développer mes propres projets.

Mais **passionné** par le Machine Learning, il n'a pas été difficile de laisser de côté les distractions pour me consacrer à mon **développement personnel**.

A travers ce guide, je veux vous offrir ce que j'ai appris car le monde a urgemment besoin de se former en Intelligence Artificielle.

Que vous souhaitiez changer de vie, de carrière, ou bien développer vos compétences à résoudre des problèmes, **ce livre vous y aidera**.

C'est **votre tour** de passer à l'action !

Ce que vous allez apprendre dans les 7 prochains jours

J'ai écrit ce livre en 7 chapitres qui retracent le cheminement naturel et logique pour apprendre le Machine Learning sans **aucun prérequis**.

Je vous invite à lire un chapitre par jour, ce qui ne vous prendra pas plus d'une demi-heure par jour.

Pour chaque chapitre, je me suis inspiré des meilleures formations qui existent à ce jour (parfois payantes) et que j'ai pu suivre (Stanford, MIT, UCL, …).

Introduction – Pourquoi devez-vous lire ce livre ?

Jour 1 : Les fondations du Machine Learning

Jour 2 : La Régression Linéaire

Jour 3 : Votre premier programme de Machine Learning

Jour 4 : La Régression Logistique et les Algorithmes de Classification

Jour 5 : Les Réseaux de Neurones

Jour 6 : Unsupervised Learning

Jour 7 : Comment gérer un projet de Machine Learning

En avant pour ce voyage qui changera peut-être votre vie comme il a pu changer la mienne !

Chapitre 1 : Les fondations du Machine Learning

Dans ce premier chapitre, nous allons voir :

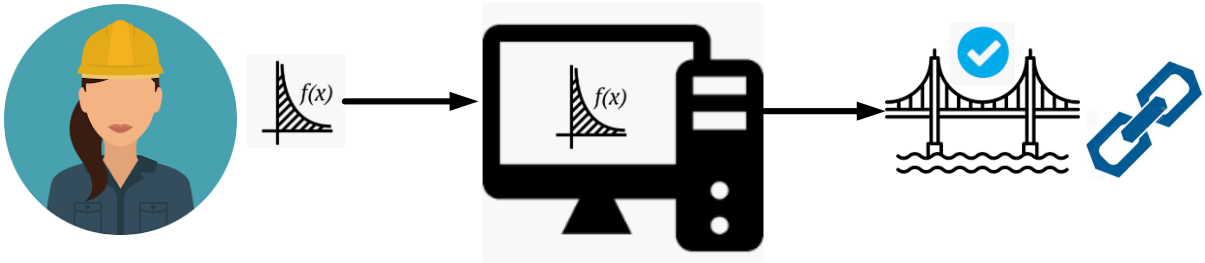
- Pourquoi le Machine Learning est vraiment utile
- La définition du Machine Learning
- Les 3 méthodes d'apprentissage
- Les 2 applications les plus courantes en Machine Learning
- Les 4 notions clefs qui s'appliquent à tout le Machine Learning

Comprendre pourquoi le Machine Learning est utilisé

Pour comprendre au mieux ce qu'est le Machine Learning et comment cela fonctionne, il faut commencer par **comprendre pourquoi** il est utilisé.

Nous, les êtres humains, sommes quotidiennement confronté à **des problèmes** que nous cherchons à **résoudre**. Par exemple : Comment construire un pont plus solide ? Comment augmenter nos bénéfices ? Comment éliminer le cancer ? Ou tout simplement quelle route emprunter pour aller au travail ?

Pour nous aider dans nos recherches, nous avons inventé **l'ordinateur**, qui permet de résoudre en quelques minutes des calculs qui nous prendraient des millions d'années à effectuer. Mais il faut savoir qu'un ordinateur ne sait en réalité faire qu'une chose : **résoudre les calculs qu'on lui donne.**



À partir de là, **2 situations** possibles :

1. On **connait** le calcul à effectuer pour résoudre notre problème.

   Dans ce cas, facile ! On entre ce calcul dans l'ordinateur, c'est ce qu'on appelle la **programmation**, et l'ordinateur nous donne le résultat.

   Exemple :
   - *Déterminer la structure d'un pont*

2. On **ne connait pas** le calcul qui résout notre problème

   Dans ce cas... on est bloqué. Impossible de donner à un ordinateur un calcul que nous ne connaissons pas.

   ***C'est comme vouloir poster une lettre que nous n'aurions pas écrite*.**

   Exemples :
   - *Reconnaitre un visage sur une photo*
   - *Prédire le cours de la Bourse*
   - *Eliminer le cancer*
   - *Composer de la musique*
   - *Conduire une voiture*

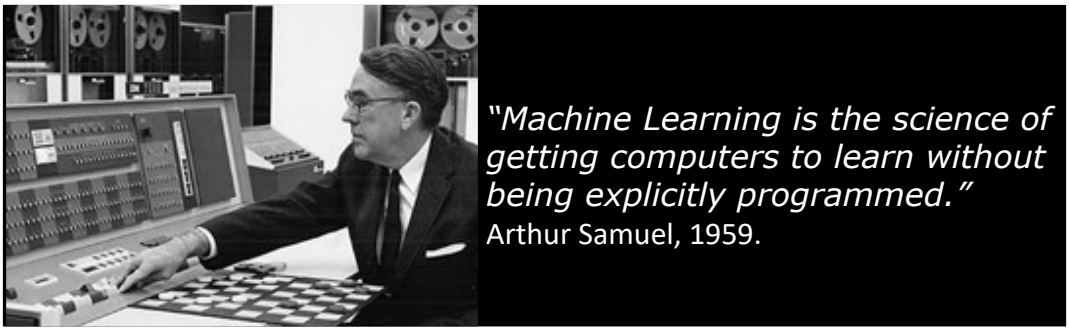
Doit-on donc perdre tout espoir de voir un jour un ordinateur nous aider dans la lutte contre le cancer ?

Bien sûr que non ! Le Machine Learning a justement été inventé pour venir **débloquer** la situation 2 (quand on ne connait pas le calcul) en utilisant une technique audacieuse, que je vais vous dévoiler tout de suite.

Laisser la Machine apprendre à partir d'expériences

Le Machine Learning consiste à laisser l'ordinateur **apprendre** quel calcul effectuer, plutôt que de lui donner ce calcul (c'est-à-dire le programmer de façon explicite).

C'est en tout cas la **définition** du Machine Learning selon son inventeur Arthur Samuel, un mathématicien américain qui a développé un programme pouvant **apprendre tout seul** comment jouer aux Dames en 1959.



Un autre américain du nom de Tom Mitchell donna en 1998 une définition un peu plus moderne du Machine Learning en énonçant qu'une machine apprend quand sa performance à faire une certaine tâche s'améliore avec de nouvelles **expériences**.

Mais comment apprendre ?

Pour donner à un ordinateur la capacité d'apprendre, on utilise des **méthodes d'apprentissage** qui sont fortement inspirées de la façon dont nous, les êtres humains, apprenons à faire des choses. Parmi ces méthodes, on compte :

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

Voyons dès à présent ce qu'est l'apprentissage supervisé, qui est la méthode la **plus utilisée** en Machine Learning.

L'Apprentissage Supervisé

Imaginez que vous commenciez à apprendre le chinois.

Pour ce faire, il vous faudra soit acheter un livre de traduction chinois-français, ou bien trouver un professeur de chinois.

Le rôle du professeur ou du livre de traduction sera de **superviser** votre apprentissage en vous fournissant des **exemples** de traductions français-chinois que vous devrez mémoriser.

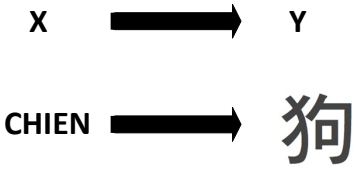
On parle ainsi **d'apprentissage supervisé** lorsque l'on fournit à une machine beaucoup **d'exemples** qu'elle doit étudier.

Pour maîtriser l'apprentissage supervisé, il faut absolument comprendre et connaitre les 4 notions suivantes :

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

Notion 1 : Apprendre à partir d'exemples (Dataset)

Comme pour apprendre la langue chinoise, on parle d'apprentissage supervisé lorsque l'on fournit à une machine beaucoup **d'exemples** $(\boldsymbol{x}, \boldsymbol{y})$ dans le but de lui faire apprendre la **relation** qui **relie** $\boldsymbol{x}$ **à** $\boldsymbol{y}$.



En Machine Learning, on compile ces **exemples** (x, y) dans un tableau que l'on appelle **Dataset** :

- La variable y porte le nom de ***target*** (la cible). C'est la valeur que l'on cherche à prédire.

- La variable x porte le nom de ***feature*** (facteur). Un facteur influence la valeur de y, et on a en général beaucoup de ***features*** $(x_1, x_2, \dots)$ dans notre Dataset que l'on regroupe dans une matrice X.

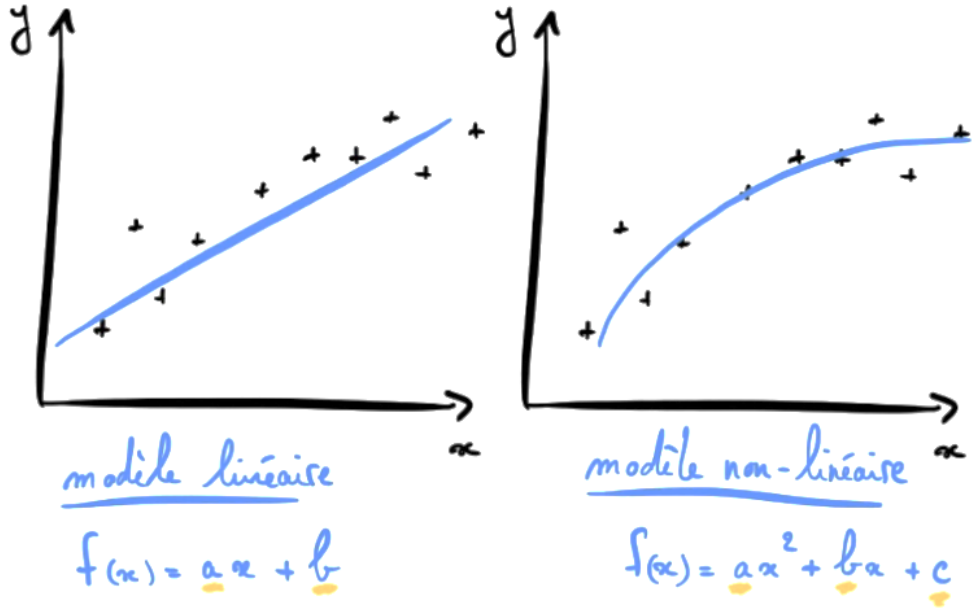
Ci-dessous, un Dataset qui regroupe des exemples d'appartements avec leur prix y ainsi que certaines de leurs caractéristiques (*features*).

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

Ce Dataset, 99.9% des gens se contentent de l'analyser dans Excel. La bonne nouvelle, c'est que vous ferez bientôt partie des **0.1%** de gens qui peuvent faire du Machine Learning avec ça !

Notion 2 : Développer un modèle à partir du Dataset

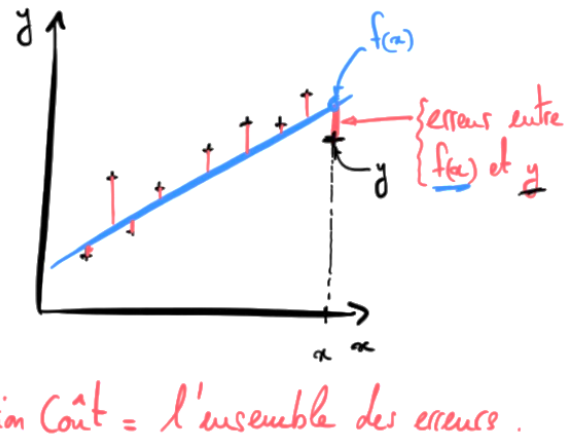
En Machine Learning, on développe un **modèle** à partir de ce Dataset. Il peut s'agir d'un modèle **linéaire** comme vous pouvez le voir à gauche, ou bien un modèle **non-linéaire** comme vous pouvez le voir à droite. Nous verrons dans ce livre comment choisir un modèle plutôt qu'un autre.



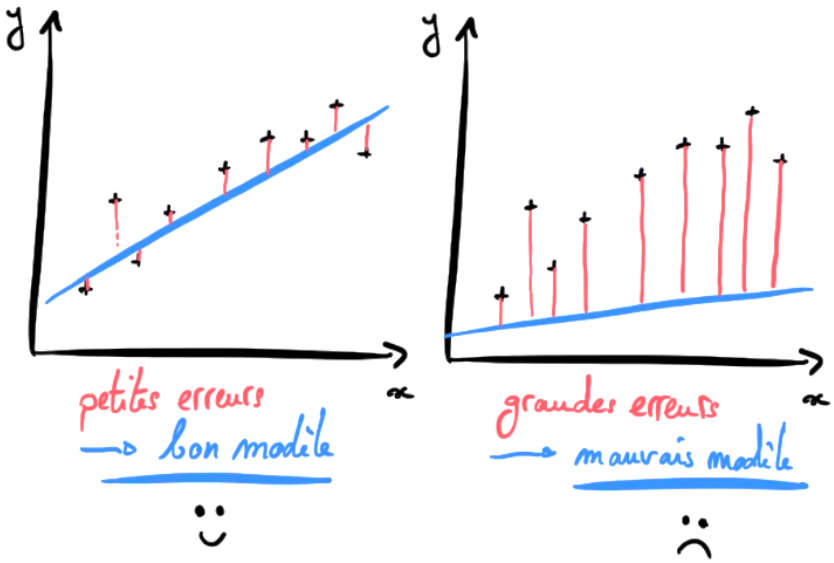
On définit a, b, c, etc. comme étant les **paramètres** d'un modèle.

Notion 3 : Les erreurs de notre modèle - la Fonction Coût

Autre chose à noter est qu'un modèle nous retourne des erreurs par rapport à notre Dataset. On appelle **Fonction Coût** l'ensemble de ces erreurs (le plus souvent on prend la moyenne quadratique des erreurs comme dans le chapitre 2).



Allons droit au but : Avoir un bon modèle, c'est avoir un modèle qui nous donne de petites erreurs, donc une **petite Fonction Coût**.



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

Ainsi l'objectif central en Supervised Learning, c'est de trouver les paramètres du modèle qui **minimisent** la Fonction Coût. Pour cela, on utilise un **algorithme d'apprentissage**, l'exemple le plus courant étant l'algorithme de **Gradient Descent**, que vous apprendrez dans le chapitre 2.

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'est-à-dire une variable qui peut prendre une **infinité** de valeurs. Par exemple :

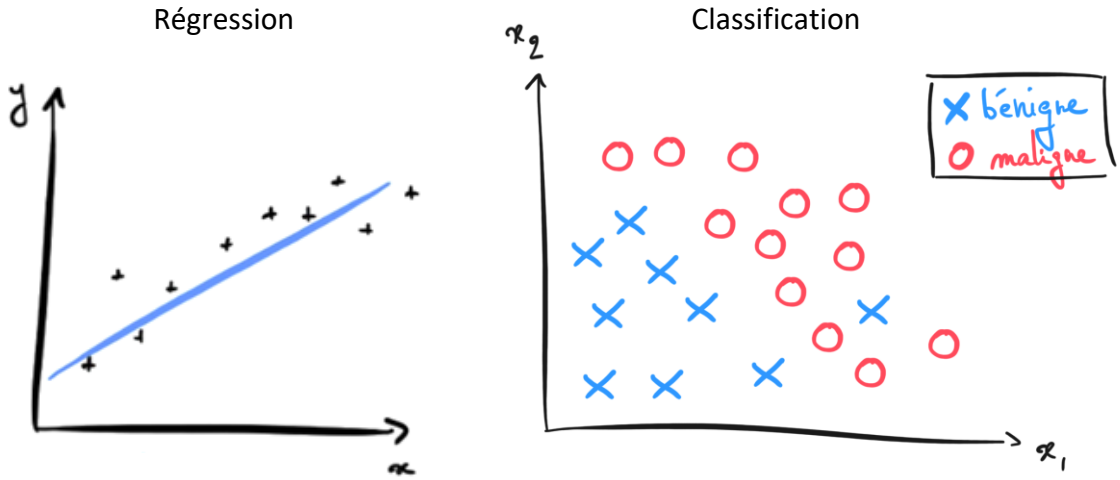
- Prédire le prix d'un appartement (y) selon sa surface habitable (x)

- Prédire la quantité d'essence consommée (y) selon la distance parcourue (x)

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

- Prédire si un email est un spam $(classe\ y\ =\ 1)$ ou non $(classe\ y\ =\ 0)$ selon le nombre de liens présent dans l'email (x)
- Prédire si une tumeur est maligne $(y = 1)$ ou bénigne $(y = 0)$ selon la taille de la tumeur $(\boldsymbol{x_1})$ et l'âge du patient $(\boldsymbol{x_2})$

Dans le cas d'un problème de classification, on représente souvent les classes par des symboles, plutôt que par leur valeur numérique (0, 1, …)



Mais tout ça, on peut le faire dans Excel ?

A ce stade, vous pourriez penser que calculer le prix d'un appartement selon sa surface habitable, tout le monde peut le faire dans **Excel** (Il existe même la fonction Régression dans Excel).

La force du Machine Learning, c'est qu'il est très facile de développer des modèles très complexes qui peuvent analyser des milliers de *features* (x) qu'un être humain ne serait **pas capable** de prendre en compte pour faire son calcul (et Excel non plus).

Par exemple, pour prédire le prix d'un appartement (y), un modèle de Machine Learning peut prendre en compte :

- sa surface $(\boldsymbol{x_1})$
- sa localisation $(\boldsymbol{x_2})$
- sa qualité $(\boldsymbol{x_3})$
- sa proximité avec un parc $(\boldsymbol{x_4})$
- etc.

De même, pour prédire si un email est un spam (y), le Machine Learning peut analyser :

- le nombre de liens $(\boldsymbol{x_1})$
- le nombre de fautes d'orthographe $(\boldsymbol{x_2})$
- etc.

Plus il y a de *features* disponibles, plus il existe d'informations pour que le modèle prenne des décisions 'intelligentes', c'est l'intelligence artificielle.

Autres méthodes d'apprentissage

Vous connaissez désormais l'apprentissage supervisé, qui s'inspire de la façon dont nous, les êtres humains, pourrions apprendre une langue comme le chinois en étudiant à l'aide d'un bouquin les associations français → chinois $(x \rightarrow y)$.

Pourtant, si vous vous perdez, seul, en Chine, sans bouquin, sans traducteur, il existe tout de même une méthode pour apprendre le chinois. C'est **l'apprentissage non-supervisé,** et je vous dévoilerai comment réussir cet exploit dans le chapitre 6.

Finalement, une 3ième méthode d'apprentissage assez populaire en robotique est **l'apprentissage par renforcement**.

Cette dernière méthode s'inspire de la façon dont nous éduquons nos animaux de compagnie, en leur offrant une friandise quand ils font une bonne action. Cette méthode étant mathématiquement plus avancée que les deux premières, je n'en parlerai pas dans ce livre, mais je vous invite à lire mon site si vous souhaitez en savoir plus !

Les 4 notions clefs du Machine Learning que vous devez absolument retenir

Le Machine Learning est un domaine vaste et complexe, et de mon expérience les gens perdent parfois de vue l'essentiel, même en suivant des formations payantes.

Pour **sortir du lot**, il faut avoir les idées claires sur les bases du Machine Learning. Vous devez ainsi retenir 4 notions essentielles, et vous verrez qu'elles vous suivront dans tous vos projets de Machine Learning.

1. Le **Dataset**

En Machine Learning, tout démarre d'un **Dataset** qui contient nos données. Dans l'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**

A 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**

Lorsqu'on teste notre modèle sur le Dataset, celui-ci nous 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 du Machine Learning, c'est de laisser la machine trouver quels sont les paramètres de notre modèle qui **minimisent** la Fonction Coût.

Chapitre 2 : La Régression Linéaire

Il est temps de mettre en pratique les concepts que vous avez appris. A travers l'exemple de la **Régression Linéaire**, vous allez mieux comprendre les notions de :

- **Dataset**
- **Modèle**
- **Fonction Coût**
- **Gradient Descent**

Petit rappel : Apprentissage Supervisé et problème de Régression

Si vous cherchez à prédire le cours de la bourse, le prix d'un appartement, ou bien l'évolution de la température sur Terre, alors vous cherchez en fait à résoudre un problème de **régression**.

Si vous disposez d'un Dataset (x, y) alors vous pouvez utiliser **l'apprentissage supervisé** pour développer un modèle de régression. Dans ce chapitre je vais vous montrer comment développer votre premier modèle de Machine Learning !

Apprenez votre premier modèle linéaire

Voici la recette à suivre pour réaliser votre premier modèle de Machine Learning.

1. Récolter vos données

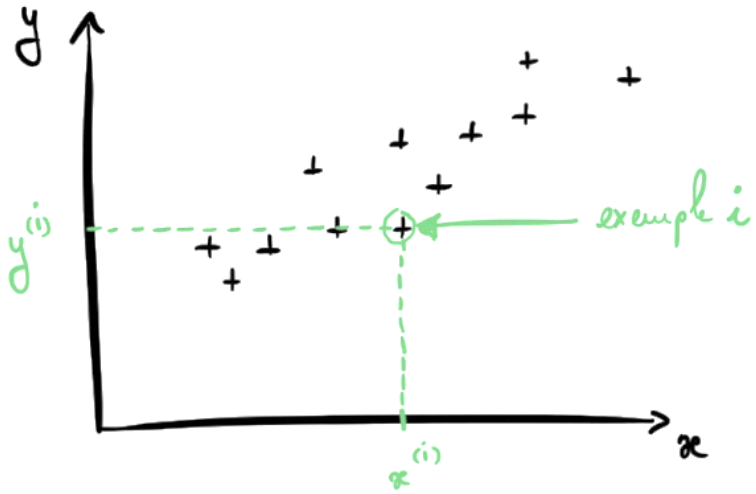
Imaginez que plusieurs agences immobilières vous aient fourni des données sur des appartements à vendre, notamment le prix de l'appartement (y) et la surface habitable (x). En Machine Learning, on dit que vous disposez de m exemples d'appartements.

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 :

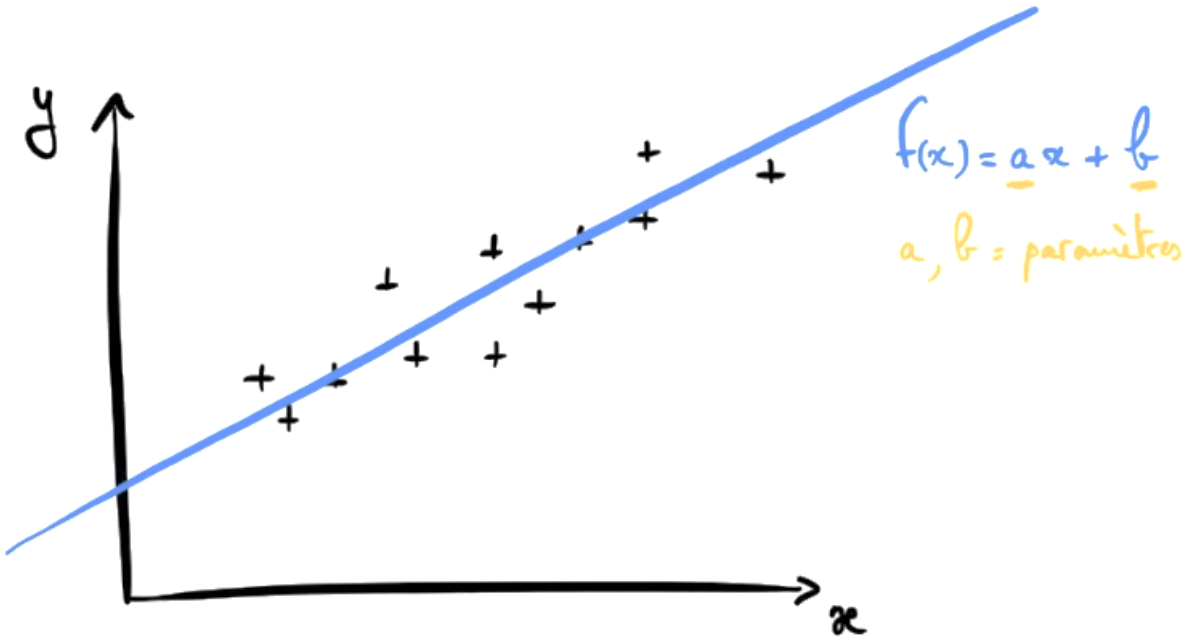


2. Créer un modèle linéaire

A partir de ces données, on développe un **modèle linéaire** $f(x) = ax + b$ où a et b sont les **paramètres** du modèle.

Un bon modèle donne 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)**.**

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

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

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

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

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

$$erreur^{(10)} = \left(2 \right)^2$$

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

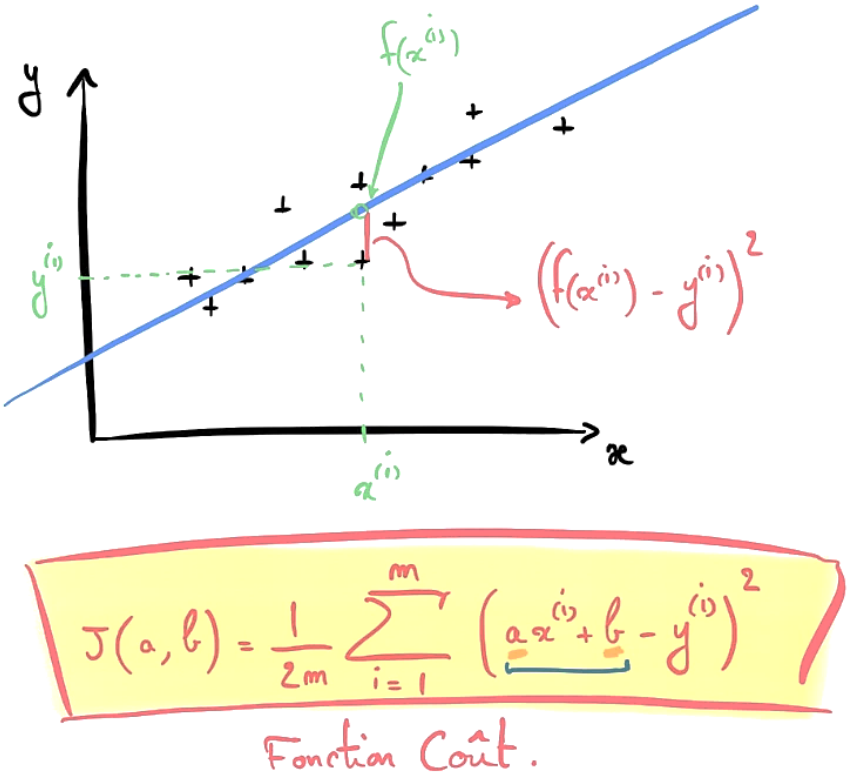
Chaque prédiction s'accompagne d'une erreur, on a donc $\boldsymbol{m}$ erreurs.

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

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

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

Note : En français, cette fonction a un nom : c'est **l'erreur quadratique moyenne (***Mean Squared Error***)**



4. Trouver les paramètres qui minimisent la Fonction Coût

La prochaine étape est l'étape la plus excitante, il s'agit de laisser la machine apprendre quels sont les paramètres qui **minimisent** la Fonction Coût, c'est-à-dire les paramètres qui nous donnent le meilleur modèle.

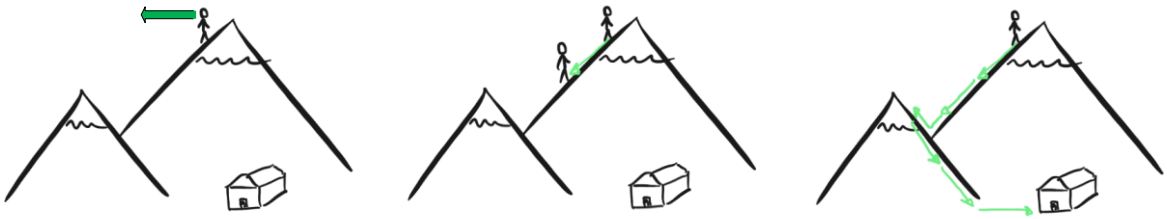
Pour trouver le minimum, on utilise un algorithme d'optimisation qui s'appelle **Gradient Descent** (la descente de gradient).

Comprendre le Gradient Descent (la descente de gradient)

Imaginez-vous perdu en montagne. Votre but est de rejoindre le refuge qui se trouve au point le **plus bas** de la vallée. Vous n'avez pas pris de carte avec vous donc vous ne connaissez pas les coordonnées de ce refuge, vous devez le trouver tout seul.

Pour vous en sortir, voici une stratégie à adopter :

1. Depuis votre position actuelle, vous partez en direction de là où la **pente descend** le plus **fort.**
2. Vous avancez une **certaine distance** en suivant cette direction coûte que coûte (même si ça implique de remonter une pente)
3. Une fois cette distance parcourue, vous répétez les 2 premières opérations en boucle, jusqu'à atteindre le point le plus bas de la vallée.



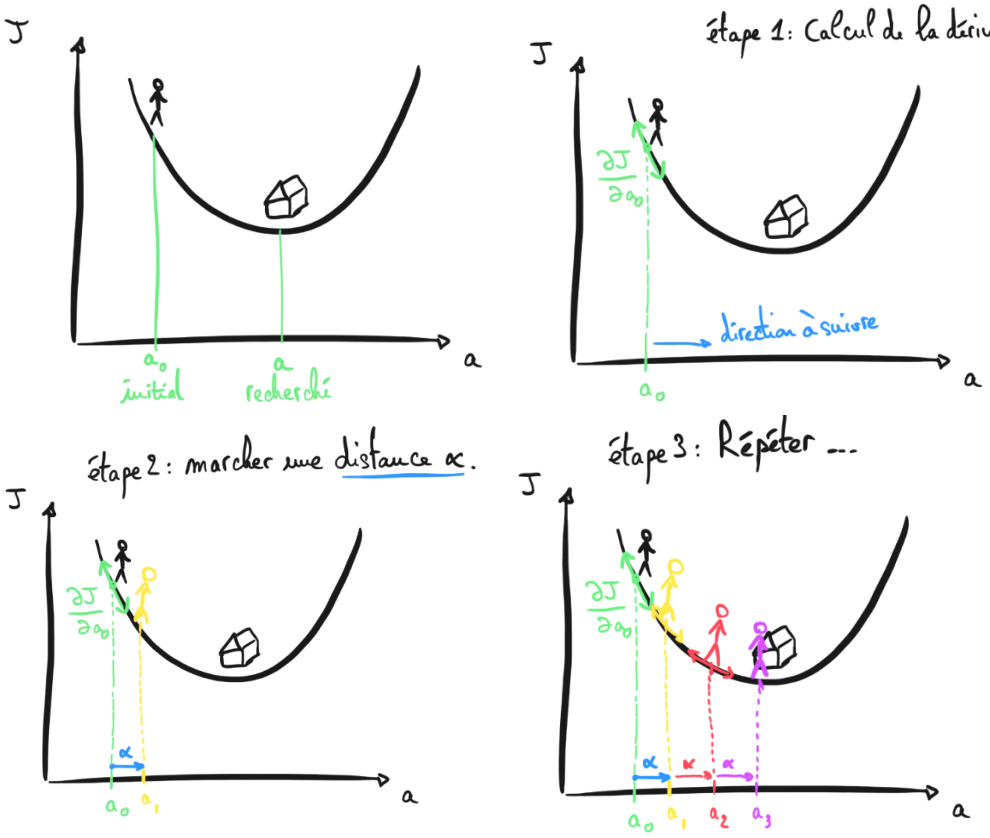
Etape 1: Trouver la pente la plus forte | **Etape 2: Marcher une certaine distance dans cette direction** | **Etape 3: Répéter les étapes 1 et 2 en boucle**

Les étapes 1, 2 et 3 forment ce qu'on appelle l'algorithme de **Gradient Descent.**

Cet algorithme vous permet de trouver le **minimum** de la Fonction Coût $J(a,b)$ (le point le plus bas de la montagne) en partant de coordonnées a et b **aléatoires** (votre position initiale dans la montagne) :

1. Calculer la **pente** de la Fonction Coût, c'est-à-dire la **dérivée** de $J(a,b)$.
2. **Evoluer** d'une certaine **distance** $\propto$ 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, voyez le dessin ci-dessous, où je montre la recherche du paramètre a idéal (la même chose s'applique au paramètre b)



Revenons à nos moutons : Comment utiliser l'algorithme de Gradient Descent

Pour rappel, nous avons jusqu'à présent créé un **Dataset**, développé un **modèle** aux **paramètres** **inconnus**, et exprimé la **Fonction Coût** $J(a,b)$ associée à ce modèle.

Notre objectif final : Trouver les paramètres a et b qui **minimisent** $J(a,b)$.

Pour cela, nous allons choisir a et b au **hasard** (nous allons nous perdre en montagne) puis allons utiliser **en boucle** la descente de gradient pour mettre à jour nos paramètres dans la direction de la Fonction Coût la **plus faible**.

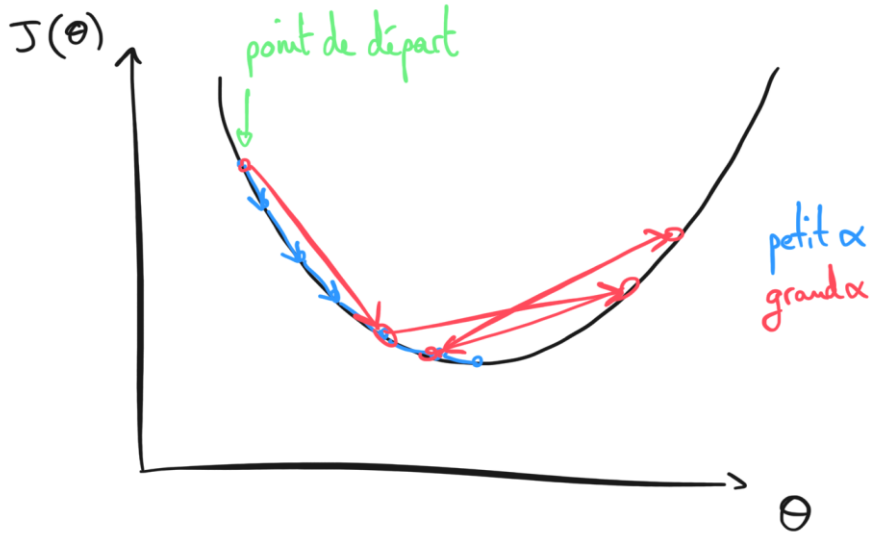
Répeter en boucle:

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

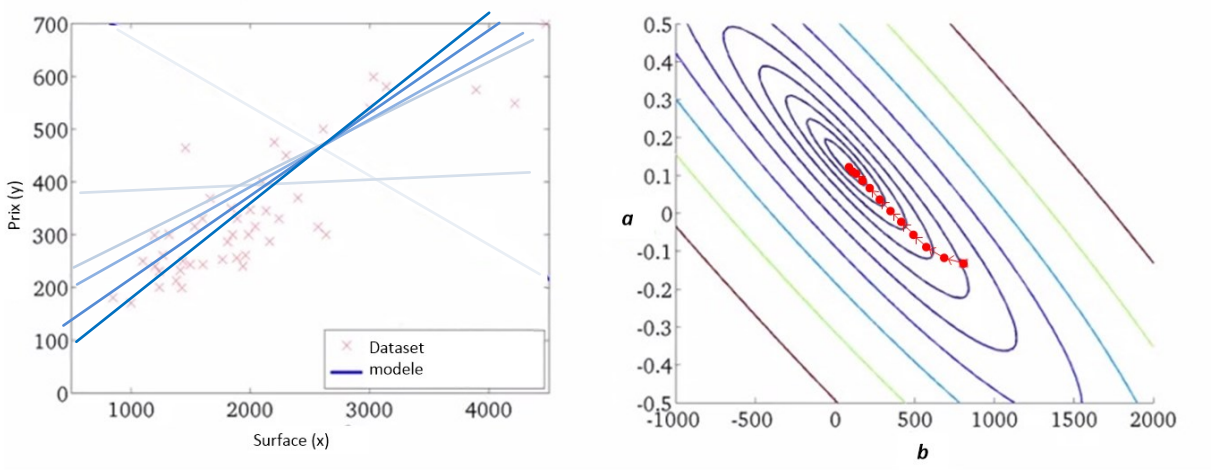
$$b = b - \propto \frac{\partial J(a,b)}{\partial b}$$

Je vous explique : à 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 $\propto$. On appelle $\propto$ la vitesse d'apprentissage (***Learning rate***).

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 peut ne jamais converger. Il est important de trouver un juste milieu. Le dessin ci-dessous illustre mes propos.



Une fois cet algorithme programmé, vous allez vivre le moment le plus excitant de votre vie de Data Scientist : voir votre première **intelligence artificielle apprendre** à prédire le prix d'un appartement selon sa surface habitable. Vous verrez comme ci-dessous que votre algorithme arrive à minimiser la Fonction Coût avec le nombre d'itérations.



A partir de là, c'est la porte ouverte aux algorithmes qui automatisent les transactions immobilières, et le même concept que celui que vous venez d'apprendre sera appliqué pour apprendre à une machine comment reconnaitre un visage sur une photo, comment prédire le cours de la bourse, etc.

Mais avant de voir la magie s'opérer, il faut avoir préalablement calculer les **dérivées partielles** de la Fonction Coût.

Calcul des dérivées partielles

Pour implémenter l'algorithme de Gradient Descent, il faut donc calculer les **dérivées partielles** de la Fonction Coût. Rappelez-vous qu'en mathématique, 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 :

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

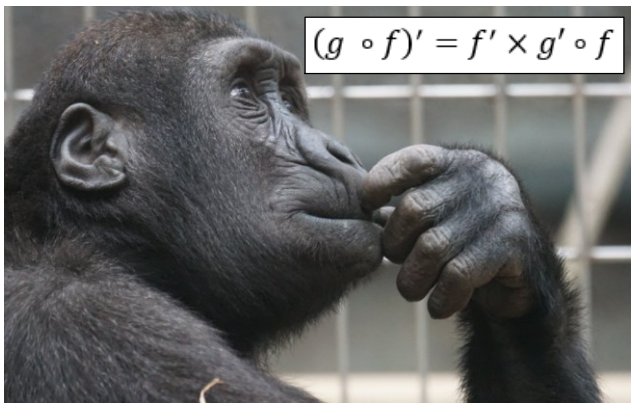
Note :

Surtout ne soyez pas impressionnés par ces formules mathématiques ! Il s'agit simplement de la dérivée d'une ***fonction composée*** *:*

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

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* $x^{(i)}$ *apparait en facteur pour la dérivée par rapport à* a.



Utilisation des matrices et des 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** $\theta = \binom{a}{b} \in \mathbb{R}^{n+1}$ 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

La recette de la régression linéaire :

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

   $Répeter\ en\ boucle$:

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

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

Le $Learning\ rate\ \alpha$ prend le nom ***d'hyper-paramètre*** de par son influence sur la performance finale du modèle (s'il est trop grand où trop petit, la fonction le Gradient Descent ne converge pas).

Dans le prochain chapitre, vous allez apprendre à programmer votre premier algorithme de Machine Learning en utilisant Python.

Chapitre 3 : Adieu Excel, bonjour Python. Vous voilà Data Scientist !

Fini la théorie, il est temps de passer à l'action !

Dans ce chapitre, vous allez apprendre à écrire de vrais programmes de Machine Learning en utilisant **Python** et le module **Sklearn.**

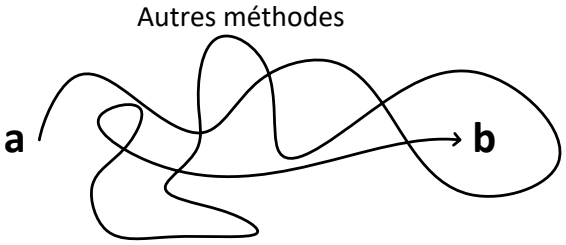
Spécifiquement pour ce chapitre, vous allez écrire un programme de **Régression Linéaire** à **plusieurs variables** et vous apprendrez comment facilement modifier votre code pour faire des **Régressions polynômiales.**

Installer Python Anaconda : le meilleur outil de Machine Learning

Il existe deux façons d'installer les outils de Machine Learning sur votre ordinateur : La bonne et la mauvaise.

En installant Anaconda, vous vous épargnez des heures de souffrance à taper des commandes en mode 'geek' à installer les packages, librairies et éditeur de texte indispensables pour faire du Machine Learning.

Pourquoi faire simple quand on peut faire compliqué ?

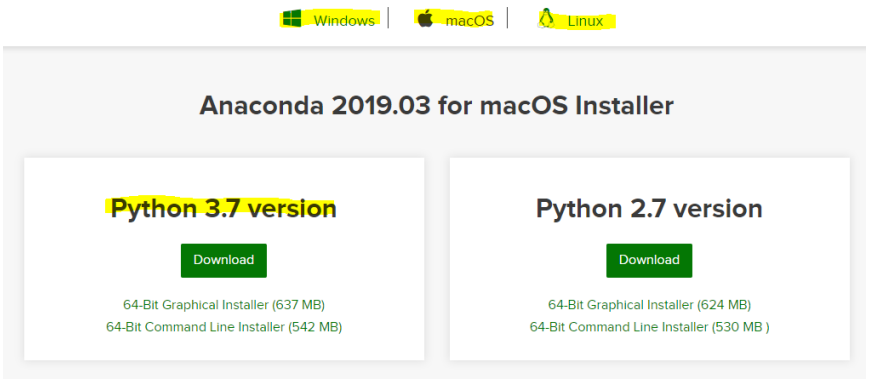


Anaconda contient tous les outils et librairies dont vous avez besoin pour faire du Machine Learning : **Numpy**, **Matplotlib**, **Sklearn**, etc.

Commencez par télécharger puis installer Anaconda depuis le site officiel :

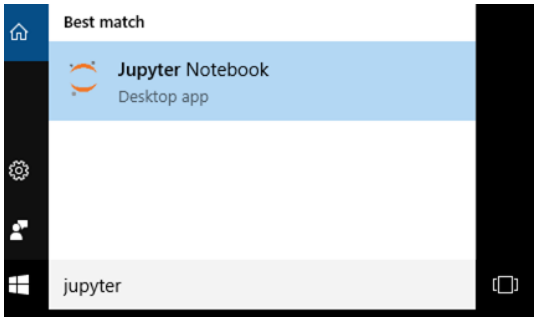
https://www.anaconda.com/distribution/#download-section

Note: Téléchargez toujours la version la plus récente de Python (ici Python 3.7)



Pour plus d'information, je vous [montre ici en vidéo] comment vous y prendre pour installer Anaconda et comment vous en servir.

Une fois Anaconda installé, vous pouvez lancer l'application **Jupyter Notebook** depuis votre barre de recherche Windows/Mac/Linux.

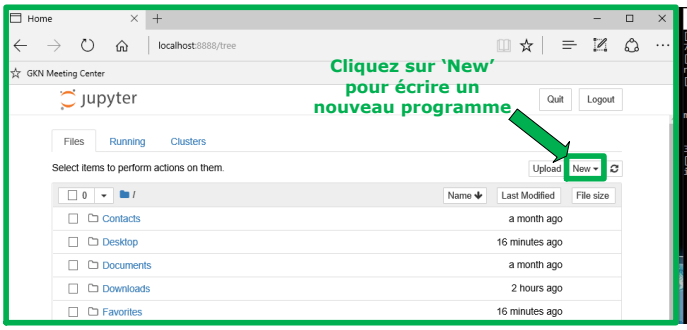


Utilisation de Jupyter Notebook pour vos projets

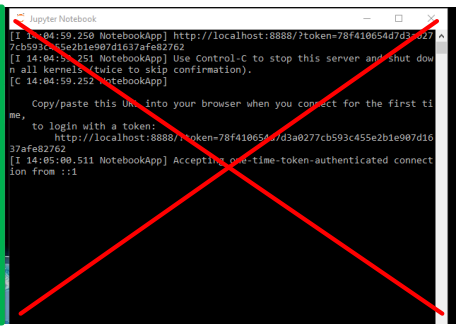
Jupyter Notebook est une application Web qui permet de créer et de partager des codes Python.

Note : C'est une application Web, mais il n'est pas nécessaire d'avoir une connexion Internet pour vous servir de Jupyter. Aussi, vos données/codes ne sont à aucun moment stockés sur Internet (ce n'est pas du Cloud).

Lorsque vous démarrez Jupyter, il est possible que 2 fenêtres s'ouvrent, auquel cas ne vous occupez pas de la fenêtre noire (la console) et surtout ne la fermez pas (ceci déconnecterait Jupyter de votre disque dur).



Fenêtre D'acceuil de Jupyter. Vous y trouvez les fichiers stockés sur votre disque dur.

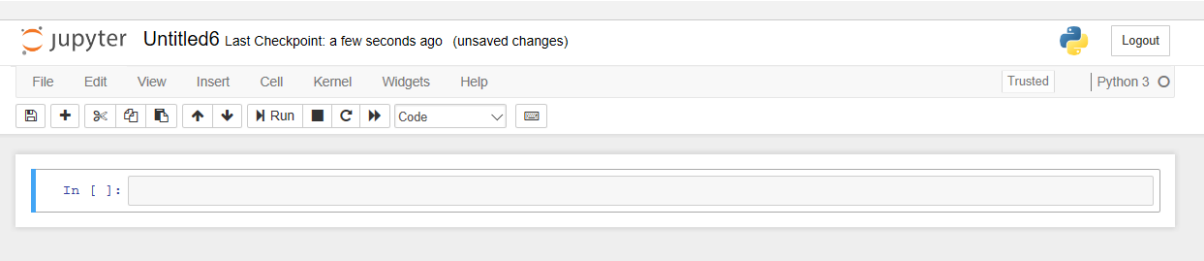


Ne vous occupez pas de cette fenêtre. Ne la fermez surtout pas!

La fenêtre principale de Jupyter n'est ni plus ni moins qu'un explorateur de fichier relié à votre disque dur, vous pouvez ouvrir tous vos fichiers, Dataset, codes, etc. depuis cette fenêtre.

Cliquez sur le bouton '*New*' situé en haut à droite de cette fenêtre pour commencer à écrire un nouveau programme (puis cliquez sur Python 3).

La fenêtre suivante s'affiche alors : Vous êtes prêts à coder !



Apprenez la programmation en 15 minutes

Il est possible que certains d'entre vous n'aient jamais écrit de programme de leur vie, alors je vais vous expliquer à ma façon comment comprendre le monde de la programmation en moins de 15 minutes.

La programmation : une nouvelle vision du monde

En programmation, on choisit d'adopter une certaine vision du monde dans lequel nous vivons. On peut considérer que le monde peut être modélisé à partir de 2 concepts :

- Des **objets**
- Des **actions**, qui modifient l'état des objets.

Regardez autour de vous. Vous êtes entourés d'objets. En réalisant une action, des objets peuvent apparaitre, disparaitre, se combiner, se transformer, etc.

D'ailleurs, quand nous nous exprimons, il est **nécessaire** et **suffisant** de dire un **nom** (objet) et un **verbe** (action) pour faire une phrase.

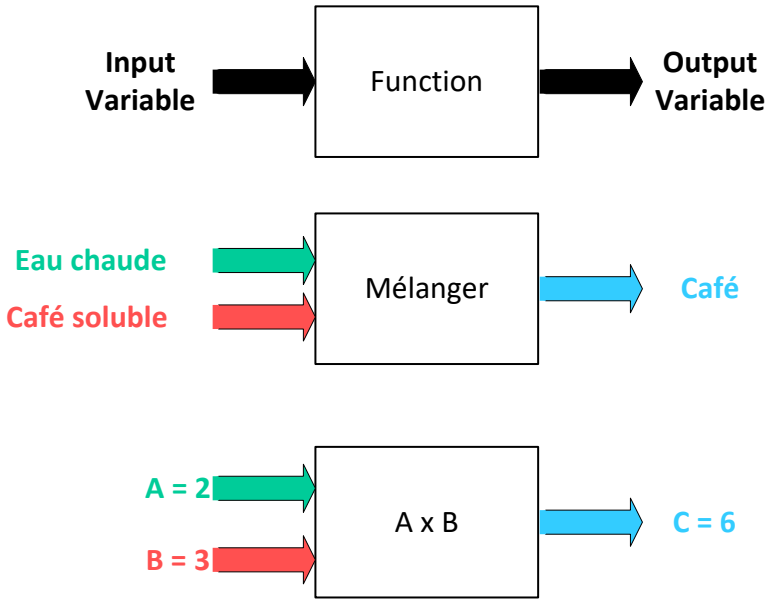
« Le chien boit. » ou bien *« Le chien boit de l'eau. »*

Chien : Objet

Boit : Action

Eau : Objet, qui diminue avec l'action boire.

En programmation, ces deux notions prennent le nom de **variable** et de **fonction,** les fonctions transformant la **valeur** des variables.



Le plus souvent, il n'est pas nécessaire de développer ses propres fonctions car celles-ci sont déjà développées dans des **librairies open source**.

Par exemple, la librairie **Sklearn** contient toutes les fonctions mathématiques et l'algorithme de Gradient Descent que nous avons appris dans le Chapitre 2 ! En pratique, il est donc inutile d'écrire la moindre équation mathématique. Génial, non ?!

Au fil des exemples dans ce livre, vous allez naturellement apprendre comment programmer en Python spécialement pour le Machine Learning. Mon but est de vous épargner une avalanche de détails inutiles que vous pourriez trouver dans une formation classique, pour vous montrer les fonctions **essentielles** et **utiles** qui vous aideront réellement à résoudre des problèmes dans votre travail après la lecture de ce livre.

Si toute fois vous désirez apprendre Python plus en profondeur, Internet regorge de formations gratuites et… ah oui j'oubliais : il y a ma chaine YouTube aussi ! ☺

Les bases de Python

1. Les commentaires

Dans tout langage de programmation, il est possible d'écrire des **commentaires.** Un commentaire n'est pas lu par la machine et vous pouvez donc y écrire ce que vous voulez pour documenter votre code.

Dans Python, il suffit de précéder votre code du symbole « # » pour le transformer en commentaire.

```
# ceci est un commentaire
```

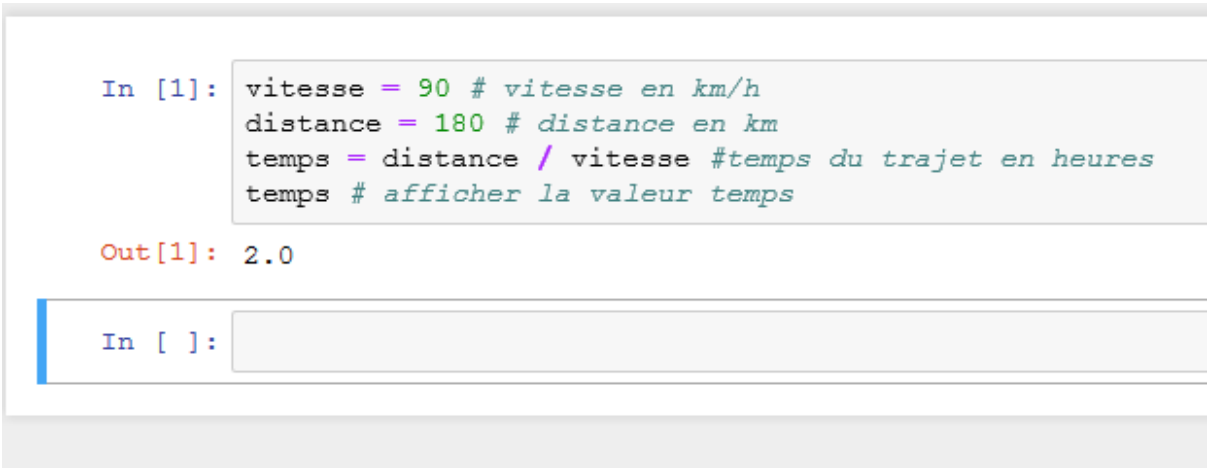
2. Les variables

Pour définir une variable dans Jupyter, il suffit de lui donner un **nom** et de lui assigner une **valeur**. Par exemple, vous pouvez choisir de créer la variable « vitesse » et de lui assigner la valeur 90.

Note : les accents et chiffres sont à bannir pour le nom d'une variable !

Vous pouvez effectuer des opérations mathématiques entre les variables numériques.

Une fois le code écrit, appuyez sur **CTRL + Entrée** pour exécuter votre code. Le résultat est affiché et une nouvelle cellule s'ouvre en bas pour continuer à écrire du code.



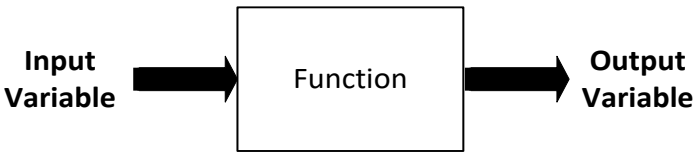
Il existe de nombreux types de variables : les variables numériques, les listes, les matrices, les chaines de caractères…

3. Les fonctions

Comme énoncé plus haut, Python contient déjà de nombreuses librairies remplies de fonctions très utiles et il n'est pas nécessaire de coder ses propres fonctions pour faire du Machine Learning.

Cependant, je vais tout de même vous montrer la structure d'une fonction dans Python, pour votre connaissance personnelle. Vous êtes libres de passer directement à la page suivante si l'envie vous prend.

Rappelez-vous qu'une fonction transforme le plus souvent une entrée en sortie :



Pour créer une fonction dans Python, il faut commencer par écrire **« def »** en début de ligne, puis donner un **nom** à la fonction, écrire les **inputs** entre **parenthèse**, et finir la ligne par un **« : »**.

Les lignes suivantes font partie de la fonction, vous pouvez y écrire des commentaires, créer de nouvelles variables, faire des opérations mathématiques etc.

La fonction s'arrête à la ligne **« return »** qui indique quelle sortie la fonction doit produire.

Une fois la fonction créée, il est possible de l'utiliser à l'infini !

Exemple :

```
In [2]: def distance_freinage(vitesse):
            # cette fonction calcul la distance d'arret en metres
            # en fonction de la vitesse a laquelle la voiture roule
            arret = vitesse * 3 / 10 #arret est une variable numérique
            return arret

In [3]: distance_freinage(45)

Out[3]: 13.5

In [ ]:
```

Voyons maintenant les principales librairies qui contiendront les fonctions à connaitre pour faire du Machine Learning **comme un pro** !

4. Les 4 librairies à maîtriser pour le Machine Learning

Pour importer une librairie dans Python, rien de plus simple. Il suffit d'écrire le nom de la librairie précédé de « *import* » en tête de votre programme. Il est également possible d'importer certaines fonctions de la librairie en écrivant `from` « *librairie* » `import` « *truc* ».

Exemple :

```
In [4]: import numpy as np
        from sklearn.datasets import make_regression

In [ ]:
```

Numpy est la librairie qui permet de créer et manipuler des **matrices** simplement et avec efficacité.

En Machine Learning, on insère le plus souvent notre **Dataset** dans des **matrices**. Ainsi le calcul matriciel représente l'essentiel du Machine Learning. Il est important de le comprendre, mais les fonctions présentes dans **Numpy** font les calculs matriciels à notre place… Magique !

Matplotlib est la librairie qui permet de **visualiser** nos Datasets, nos fonctions, nos résultats sous forme de graphes, courbes et nuages de points.

Sklearn est la librairie qui contient toutes les fonctions de l'état de l'art du **Machine Learning.** On y trouve les **algorithmes** les plus **importants** ainsi que diverses fonctions de ***pre-processing***.

Pandas est une excellente librairie pour **importer** vos tableaux Excel (et autres formats) dans Python dans le but de tirer des **statistiques** et de charger votre Dataset dans Sklearn.

5. Et tout ce dont je n'ai pas parlé

Dans cette introduction à Python, je n'ai pas parlé des boucles ***for*** et ***while.*** Je n'ai pas parlé des conditions ***if elif else*** et j'ai omis d'introduire d'autres commandes de bases comme « *print('hello world') ».*

Ces codes sont bien évidemment **importants**, mais ils ne sont pas essentiels à la compréhension et à l'apprentissage du Machine Learning pour ce livre.

Je vous invite à consulter ma chaine YouTube si vous désirez compléter vos bases en Python. ☺

Développer enfin votre premier programme de Machine Learning

Fini de rigoler, il est temps de passer à l'action ! Vous allez maintenant développer un programme de **régression linéaire** en suivant la méthode apprise dans le **chapitre 2**.

Les étapes pour programmer une Régression Linéaire

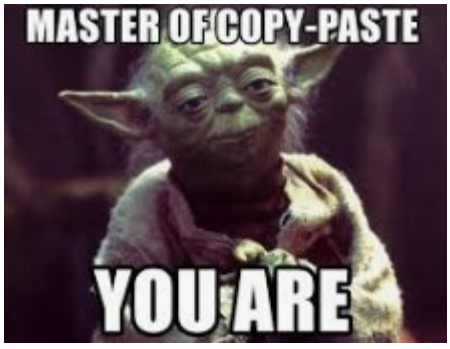
Etape 1 : Importer les librairies

Commençons par ouvrir un nouveau Notebook dans Jupyter comme nous l'avons appris précédemment. Ensuite, il faut importer les librairies et fonctions suivantes :

- **Numpy** pour manipuler notre Dataset en tant que matrice
- **Matplotlib.pyplot** pour visualiser nos données
- La fonction **make_regression** de **Sklearn** pour générer un nuage de point (ici on va simuler des données)
- **SGDRegressor** (qui signifie Stochastic **Gradient Descent** Regressor) et qui contient le calcul de la **Fonction Coût,** des **gradients**, de **l'algorithme** de **minimisation**, bref… tout ce qui pouvait sembler compliqué dans le chapitre 2.

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

N'oubliez pas de taper **CTRL + Entrée** pour exécuter votre code. S'il y a une erreur, réviser vos compétences du copier/coller… ☺



Etape 2 : Créer un Dataset

Pour ce premier code, nous n'allons pas importer de données personnelles. Plutôt, nous allons générer un tableau de données $(\boldsymbol{x}, \boldsymbol{y})$ aléatoires.

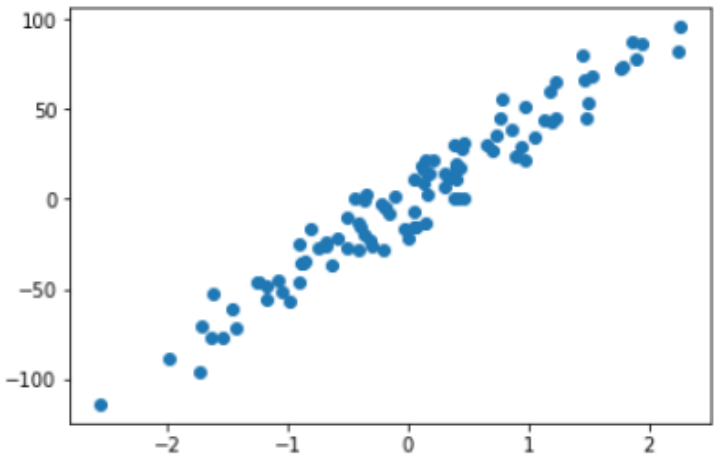
Pour cela, la fonction `make_regression` est très utile. La fonction prend comme **arguments** (c'est le mot pour désigner inputs) le nombre d'échantillons à générer, le nombre de variables et le bruit puis nous retourne deux vecteurs $\boldsymbol{x}$ et $\boldsymbol{y}$.

Pour maitriser l'aléatoire, on écrit la ligne `np.random.seed(0)`.

Finalement, pour visualiser nos données on utilise la fonction `plt.scatter(x, y)`.

```
np.random.seed(0)
x, y = make_regression(n_samples=100, n_features=1, noise=10)
plt.scatter(x, y)
```

Voici le résultat que vous devriez obtenir :



Etape 3 : Développer le modèle et l'entraîner

Pour développer et entraîner un modèle, il a fallu beaucoup de maths dans le chapitre 2 : Entre la Fonction Coût, les dérivées, l'algorithme de Gradient Descent…

Dans **Sklearn**, tout cela est **déjà fait** pour vous !

Il vous suffit de définir une variable *model* depuis le générateur **SGDRegressor** en entrant le nombre d'itérations que le Gradient Descent doit effectuer ainsi que le Learning Rate.

Une fois le modèle défini, il vous faut **l'entraîner.** Pour cela, il suffit d'utiliser la fonction **fit**.

Par exemple, entraînons notre modèle sur **100** itérations avec un Learning rate de **0.0001** :

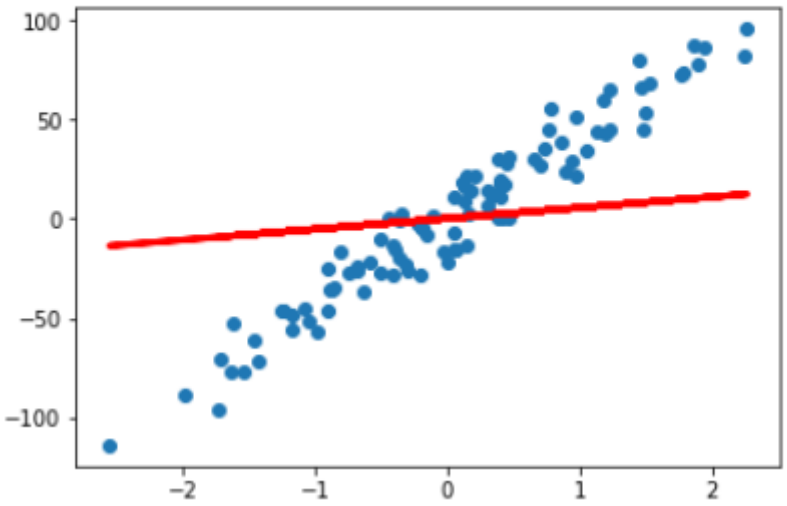
```
model = SGDRegressor(max_iter=100, eta0=0.0001)
model.fit(x,y)
```

Nous pouvons maintenant observez la précision de notre modèle en utilisant la fonction **score** qui calcule le **coefficient de détermination** entre le modèle et les valeurs y de notre Dataset.

On peut aussi utiliser notre modèle pour faire de nouvelles prédictions avec la fonction **predict** et tracer ces résultats avec la fonction **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>]
```

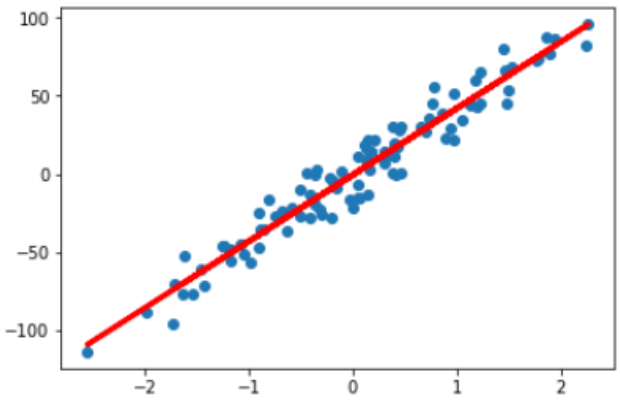


Wow ! Notre modèle semble vraiment mauvais. C'est parce que nous ne l'avons pas entraîné suffisamment **longtemps** et parce que le *Learning rate* était trop **faible**. Aucun problème, il est possible de le ré-entraîner avec de meilleurs hyper-paramètres.

En Machine Learning, les valeurs **qui fonctionnent bien** pour la plupart des entraînements sont :

- Nombre d'itérations = **1000**
- Learning rate = **0.001**

```
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>]
```



Fantastico ! Vous avez entraîné votre premier modèle de Machine Learning, et il fonctionne vraiment bien avec un coefficient $R^2 = 94\%$. Vous pourriez maintenant vous en servir pour faire de bonnes prédictions ! Par exemple pour prédire le prix d'un appartement selon sa surface habitable, ou bien pour prédire l'évolution de la température sur Terre.

Mais peut-être n'êtes-vous pas très satisfait… On peut aussi faire ce genre de chose dans Excel, pourquoi se compliquer la vie ?

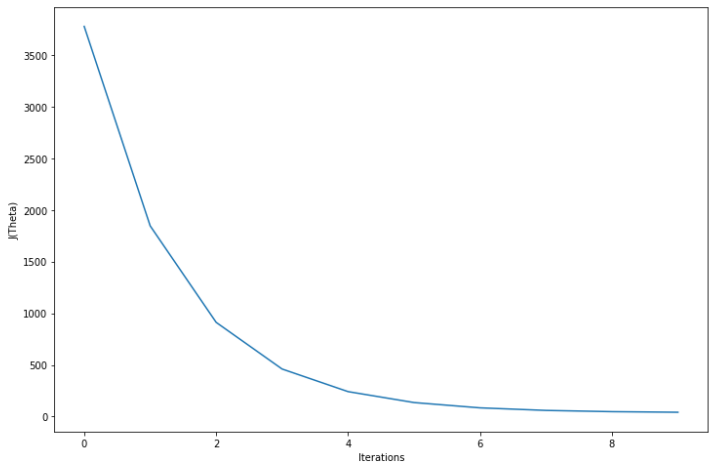
Certes, mais les choses vont commencer à devenir plus excitantes dans les prochaines pages quand nous allons développer des modèles à partir de **centaines** de **variables** (ici nous n'en avions qu'une : x)

Mais auparavant, je vais vous montrer comment votre machine a appris les paramètres du modèle avec le Gradient Descent. Pour cela, il existe ce qu'on appelle les **courbes d'apprentissage**.

Les courbes d'apprentissage

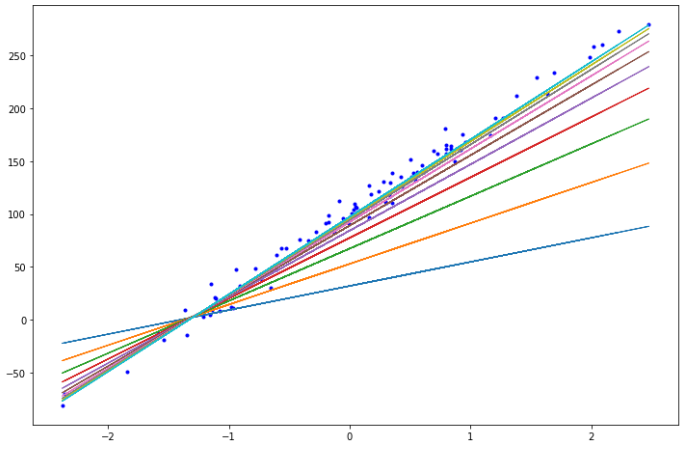
En Machine Learning, on appelle courbe d'apprentissage (***Learning curves***) les courbes qui montrent l'évolution de la Fonction Coût au fil des itérations de Gradient Descent. Si votre modèle apprend, alors sa Fonction Coût doit diminuer avec le temps, comme ci-dessous :

```
|: fig,ax = plt.subplots(figsize=(12,8))

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



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

```
In [20]: 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)
```



Si vous souhaitez reproduire ces courbes, je vous ai mis à la fin de ce livre le code que vous pourrez copier/coller (en bonus je vous donne la version longue du code de Gradient Descent telle que nous l'avons vu dans le chapitre 2).

Régression Polynômiale à plusieurs variables

Si vous achetez un stylo à 1€, combien vous coûteront 100 stylos ?

100 € ? **Faux !**

Nous vivons dans un monde régit par des lois souvent **non**-**linéaires** et où une infinité de facteurs peuvent influencer nos résultats.

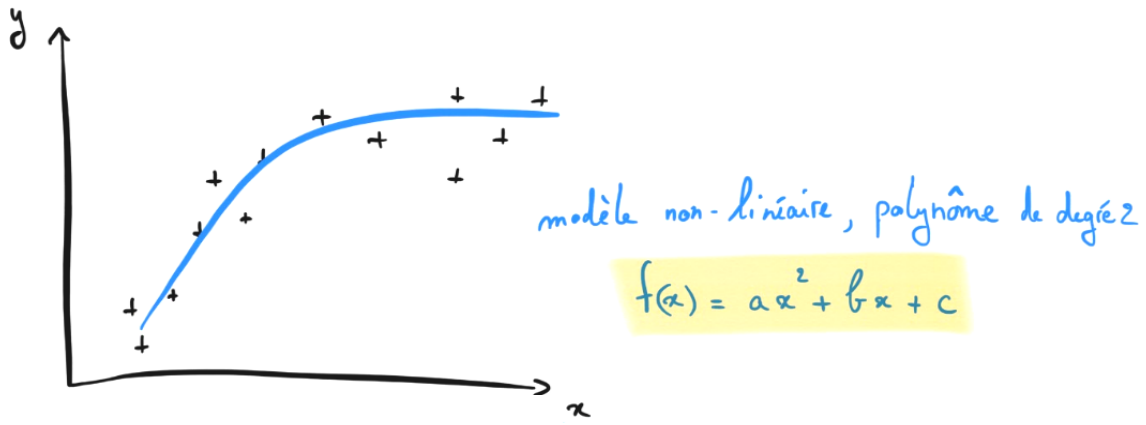
Par exemple, si vous achetez 100 stylos, vous aurez peut-être une réduction à 90 €. Si en revanche il y a une pénurie de stylos, ce même stylo qui coûtait 1 € pourrait valoir 1.50 €

C'est là qu'Excel ne pourra plus rien pour vous et que le Machine Learning trouve son utilité dans le monde réel.

Problème non-linéaire : Un problème plus compliqué ?

Pour le nuage de point ci-dessous, il semblerait judicieux de développer un modèle **polynômial** de **degré 2**.

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



Ce modèle, plus **complexe** que le modèle linéaire précédent, va engendrer des calculs algébriques plus intenses, notamment le calcul des dérivées … **ou pas !**

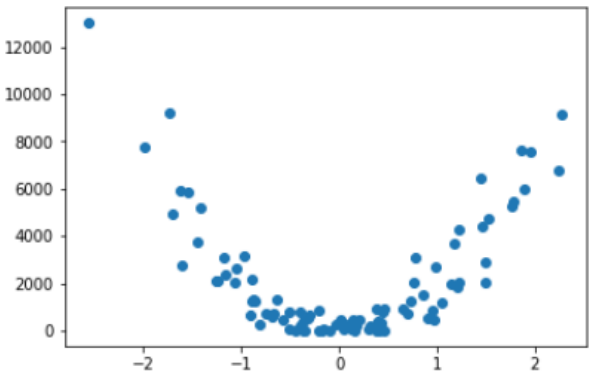
En fait, le code que nous avons écrit pour la régression linéaire peut être utilisé pour des problèmes bien plus complexes. Il suffit de générer des **variables polynômiales** dans notre Dataset en utilisant la fonction **`PolynomialFeatures`** présente dans Sklearn.

```
from sklearn.preprocessing import PolynomialFeatures
```

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

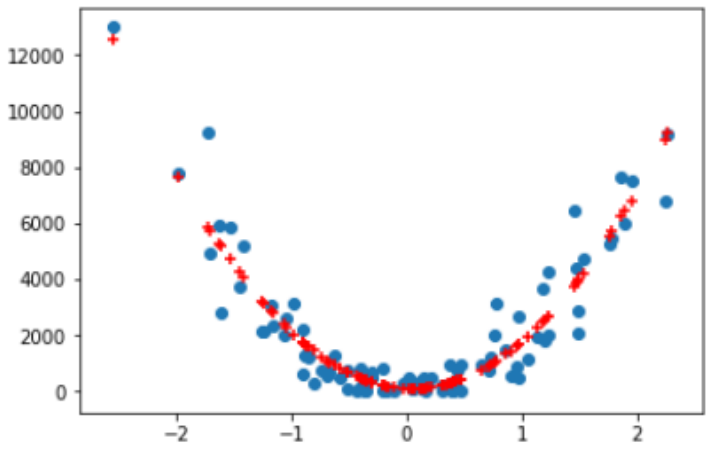
Dans l'exemple ci-dessous, j'ai choisi d'ajouter une variable polynômiale de **degré 2** pour forcer la machine à développer un modèle qui épousera l'allure **parabolique** de y en fonction de x.

```
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 polynômiales 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 entraine le modele 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 capable de prédire des résultats sur des milliers de dimensions (un exemple ci-dessous)

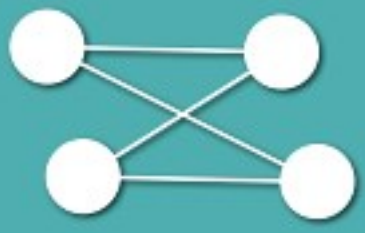
Résumé de ce Chapitre

Pour faire vos premiers pas dans le Machine Learning, vous avez installé **Anaconda** Python qui comprend tous les outils et librairies nécessaires (Jupyter, Numpy, Sklearn etc).

Avec **Sklearn**, il suffit d'écrire quelques lignes pour développer des modèles de Régression Linéaire et Polynômiale. Vous devez vous souvenir des fonctions suivantes

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

Je n'ai pas parlé de la fonction de `Sklearn.linear_model.LinearRegression` car cette méthode **n'intègre pas** l'algorithme de Gradient Descent. Elle repose en fait sur les **Equations Normales**, et fonctionne **très bien**, mais s'adapte mal aux gros Datasets (quand il y a plusieurs centaines de *features*). Nous utiliserons cependant cette méthode dans le Chapitre 7.



Chapitre 4 : Régression Logistique et Algorithmes de Classification

Dans l'apprentissage supervisé, il y a deux type de problèmes :

- Les régressions
- Les classifications

Dans ce chapitre, vous allez découvrir le modèle de Régression Logistique, qui permet de résoudre des problèmes de **classification binaires**.

Je vais aussi vous présenter un des algorithmes les plus populaires et simple : Le **K-Nearest Neighbour.**

Les problèmes de Classification

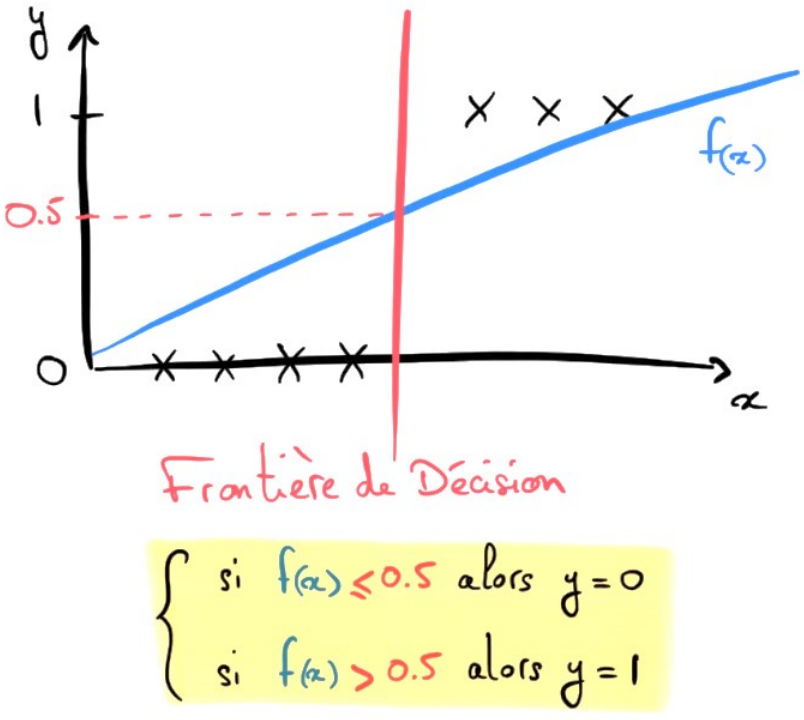
Jusqu'à présent, nous avons appris comment résoudre des problèmes de régression. Au cours du chapitre 1, j'ai parlé des problèmes de **classification**, qui consistent par exemple à classer un email en tant que 'spam' ou 'non spam'.

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

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

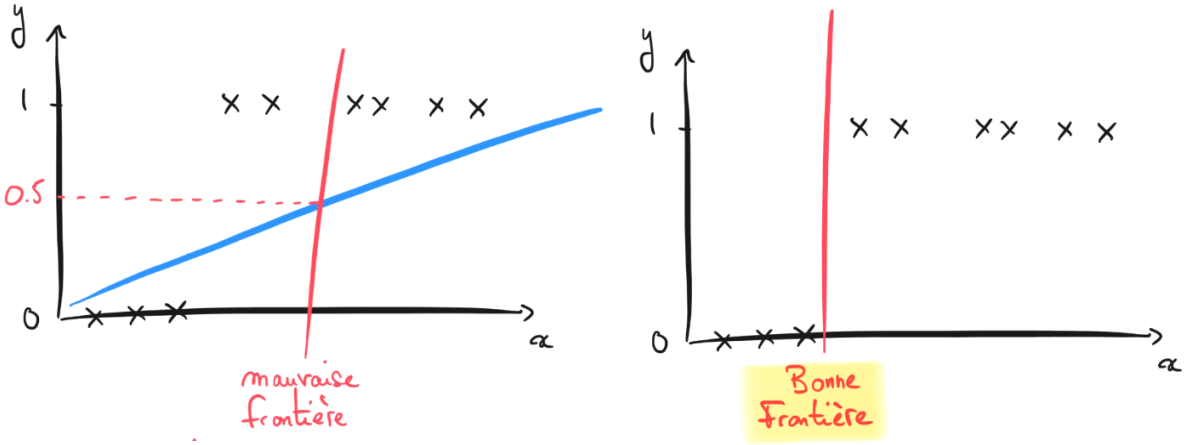
On dit également que l'on a **2 classes**, c'est une classification **binaire**.

Pour ces problèmes, on ajoute au modèle une **frontière de décision** qui permet de classer un email dans la $classe\ 0$ ou la $classe\ 1$.

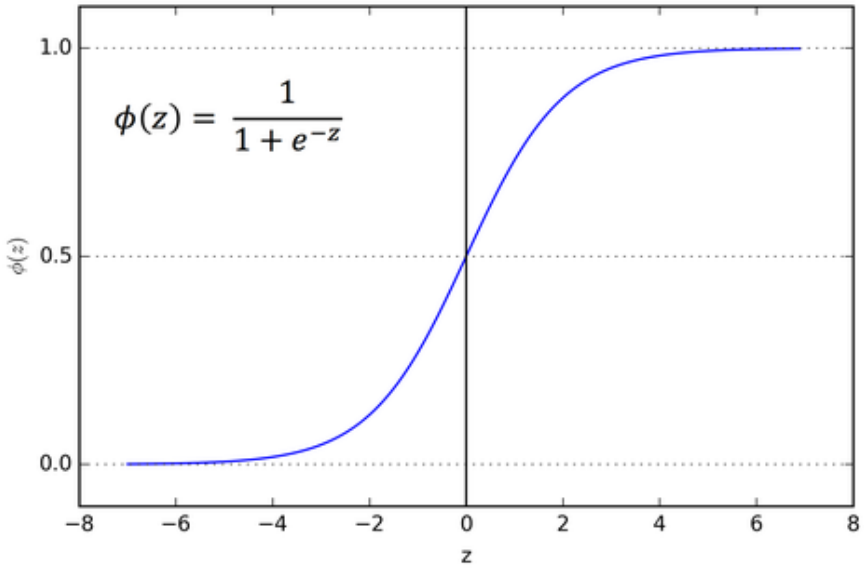


Le modèle de Régression logistique

Pour les problèmes de classification binaire, un modèle linéaire $\boldsymbol{F = X.\theta}$, comme je l'ai tracé sur la figure précédente, ne convient pas. Voyez plutôt le résultat que l'on obtient avec un tel modèle pour le Dataset suivant :



On développe alors une nouvelle fonction pour les problèmes de classification binaire, c'est la **fonction logistique** (aussi appelé fonction **sigmoïde** ou tout simplement sigma $\boldsymbol{\sigma}$). Cette fonction a la particularité d'être toujours comprise en 0 et 1.

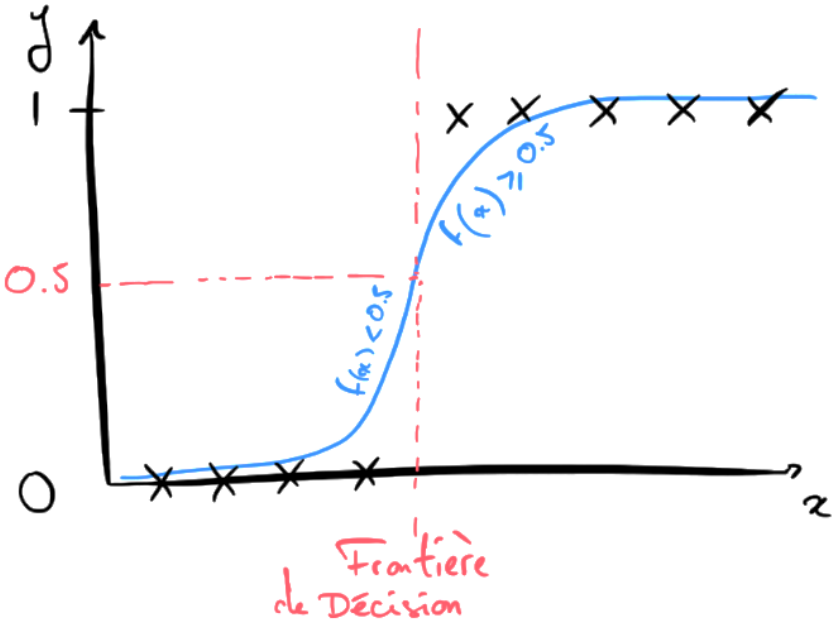


Pour coller la fonction logistique sur un Dataset $(\boldsymbol{X}, \boldsymbol{y})$ on y fait passer le produit matriciel $\boldsymbol{X.\theta}$ ce qui nous donne le **modèle** de *Logistic Regression* :

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

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

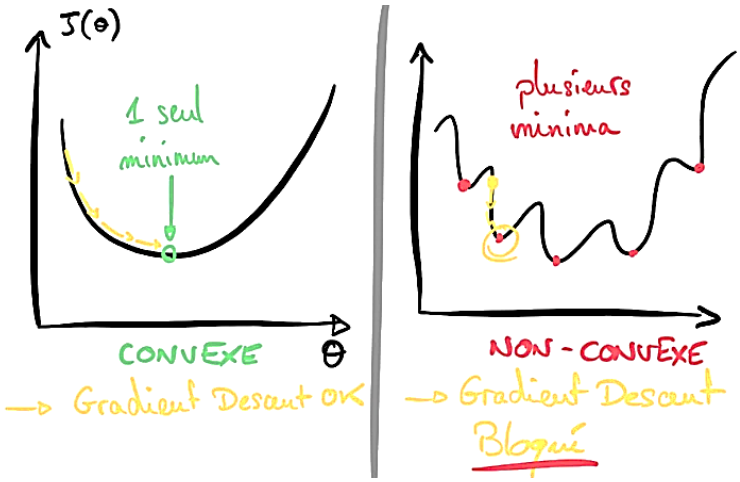
$$\begin{cases} \boldsymbol{y = 0\ si\ \sigma(X.\theta) < 0.5} \\ \boldsymbol{y = 1\ si\ \sigma(X.\theta) \geq 0.5} \end{cases}$$



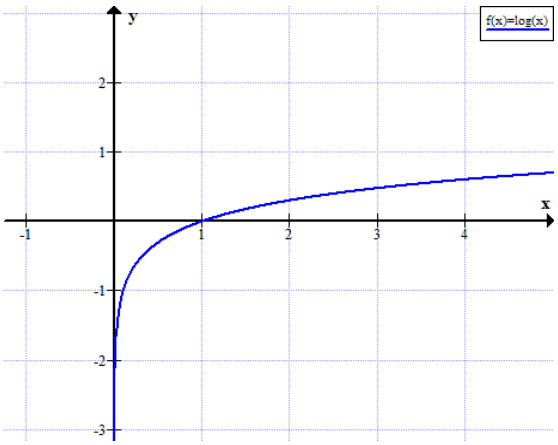
Fonction Coût associée à la Régression Logistique

Pour la régression linéaire, la Fonction Coût $\boldsymbol{J(\theta) = \frac{1}{2m}\sum(X.\theta - Y)^2}$ donnait une courbe **convexe** (qui présente un unique minima). C'est ce qui fait que l'algorithme de Gradient Descent fonctionne.

En revanche, 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** au premier minima rencontré, sans trouver le minimum **global**.



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



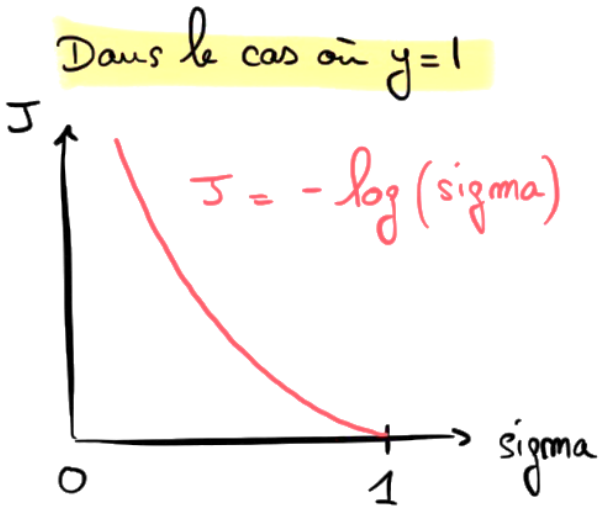
Fonction Coût dans les cas où $y = 1$

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

$$\boldsymbol{J(\theta) = -\log\left(\sigma(X.\theta)\right)}$$

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 les **paramètres** $\boldsymbol{\theta}$ pour un coût qui tend vers 0.



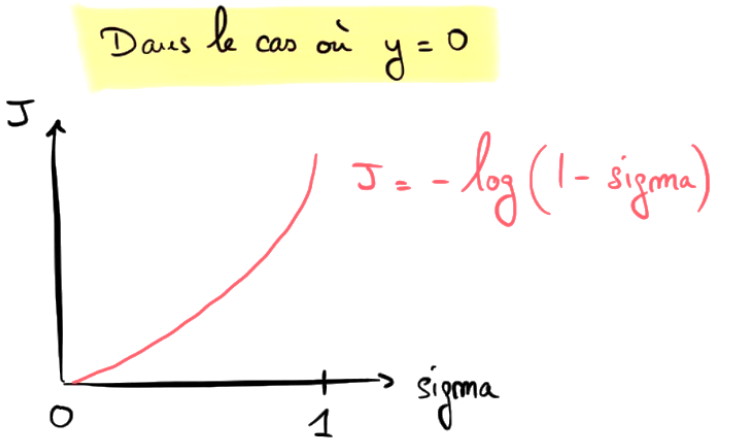
Fonction Coût dans les cas où $y = 0$

Cette fois la Fonction Coût devient :

$$\boldsymbol{J(\theta) = -\log\left(1 - \sigma(X.\theta)\right)}$$

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 $\boldsymbol{-\log(1-0)}$ donne la même courbe, 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 :

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

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

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

Et dans le cas où $y = 1$

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

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 est la même aussi ! On a :

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

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

Résumé de la Régression Logistique

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

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

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

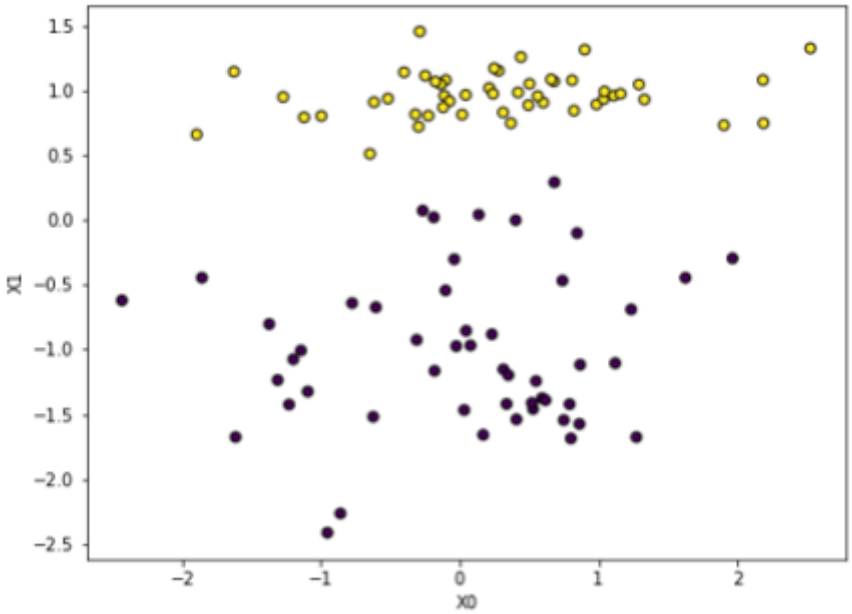
$$\textbf{Gradient Descent}: \theta = \theta - \alpha \times \frac{\partial J(\theta)}{\partial \theta}$$

Développer un programme de classification binaire dans Jupyter

On est reparti dans Jupyter ! Comme pour le chapitre 3, nous allons générer des données aléatoires, mais cette fois-ci avec la fonction make_classification. Commençons par importer nos modules habituels :

```
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
```

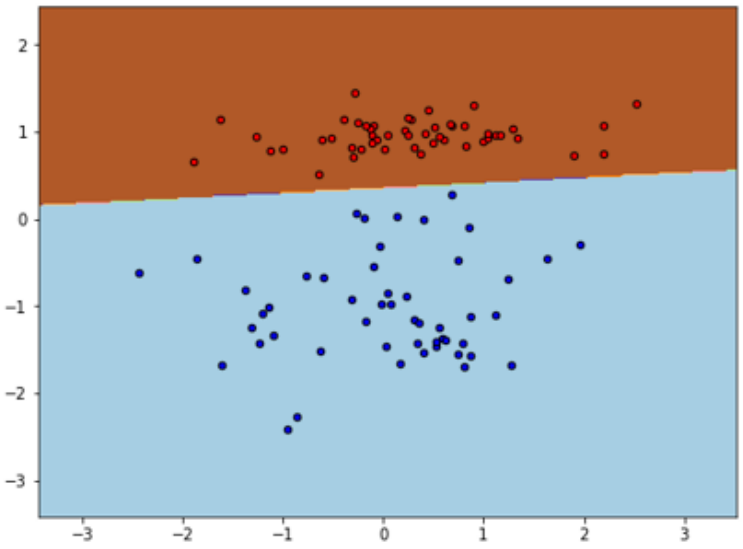


Ensuite, nous devons créer un modèle en utilisant **`SGDClassifier`**.

```
# Génération d'un modele 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 avec le code suivant…un peu compliqué je vous l'accorde, mais un simple copier/coller fera l'affaire, pas vrai ? ☺

```
# 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)
```



Vous pouvez désormais réutiliser ce code sur vos propres données pour ainsi prédire si un email est un spam ou encore si une tumeur est maligne ou non.

Bon. Nous avons vu jusqu'à présent des algorithmes bourrés de maths et donc pas forcément *fun* à étudier… mais ça va changer tout de suite !

L'Algorithme de Nearest Neighbour

Je vais désormais vous montrer l'algorithme qui est probablement le plus **simple** à comprendre de tous !

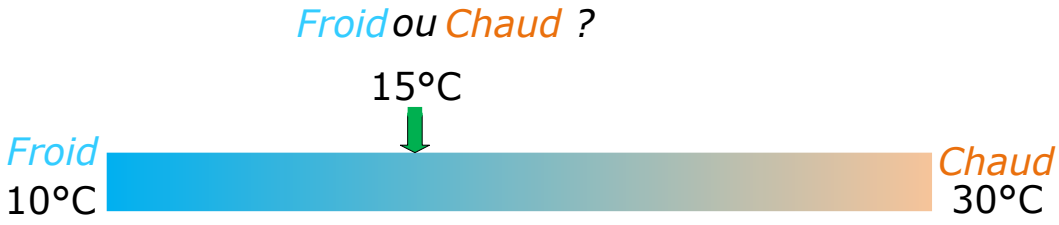
L'algorithme de **Nearest Neighbour** (le voisin le plus proche) permet de résoudre des problèmes de classification à **plusieurs classes** de façon simple et très efficace.

Promenade en Montage

Vous partez vous promener en montagne avec un ami. Avant de partir, il fait 30 °C et votre ami vous dit qu'il a chaud. Arrivé en montagne, il fait désormais 10 °C et votre ami vous dit qu'il a froid.

En redescendant la vallée, il fait maintenant 15 °C, pensez-vous que votre ami aura froid ou bien chaud ?

15 °C étant plus proche de 10 °C (froid) que de 30 °C (chaud), il semble légitime de prédire que votre ami aura froid.



Voilà l'essentiel de ce qu'il y à savoir sur l'algorithme **Nearest Neighbour**. Quand vous devez faire une nouvelle prédiction, trouvez dans votre **Dataset** l'exemple le plus **proche** par rapport aux **conditions** dans lesquelles vous êtes.

Eh ! Qui a dit que le Machine Learning était difficile ?

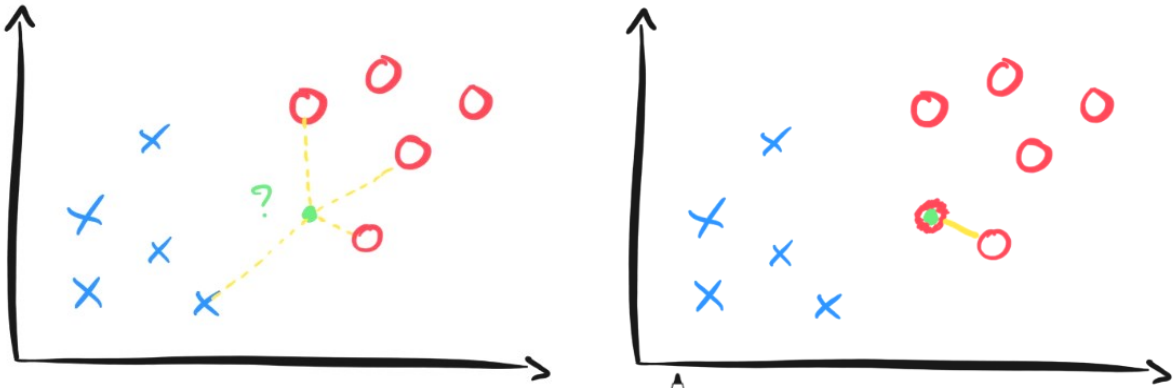
Note :

Cet exemple vous montre au passage que la **variété** et la **quantité** de données dans votre Dataset est primordiale ! Si vous ne disposez que de 2 points : -20 °C = froid ; 20 °C = chaud, alors vous pourriez conclure que 1 °C est une température chaude… Pas sûr que ça plaise à tout le monde ! On en reparlera dans le Chapitre 7.

K-Nearest Neighbour (K-NN)

La distance la plus courte

Regardez le nuage de points qui suit. Quel est le l'exemple le plus proche du point vert ? C'est un exemple de la 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.



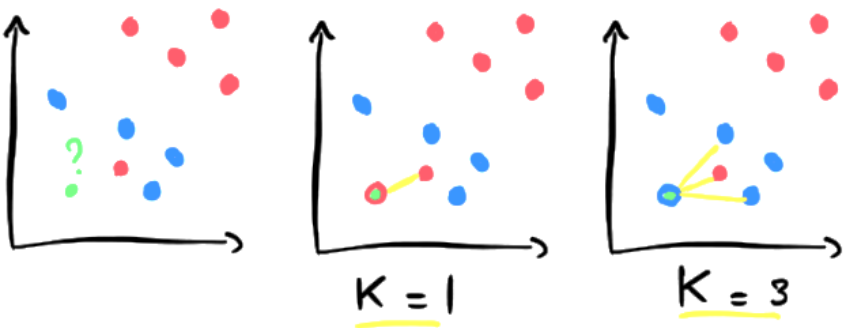
Typiquement, on utilise la **distance euclidienne** (c'est la droite direct entre deux points) mais d'autres métriques sont parfois plus utiles, comme la distance de **Manhattan** ou bien la distance **cosinus**.

Inutile de rentrer dans les détails mathématiques, vous savez désormais que **Sklearn** implémente toutes les équations pour vous.

Le nombre de voisin K

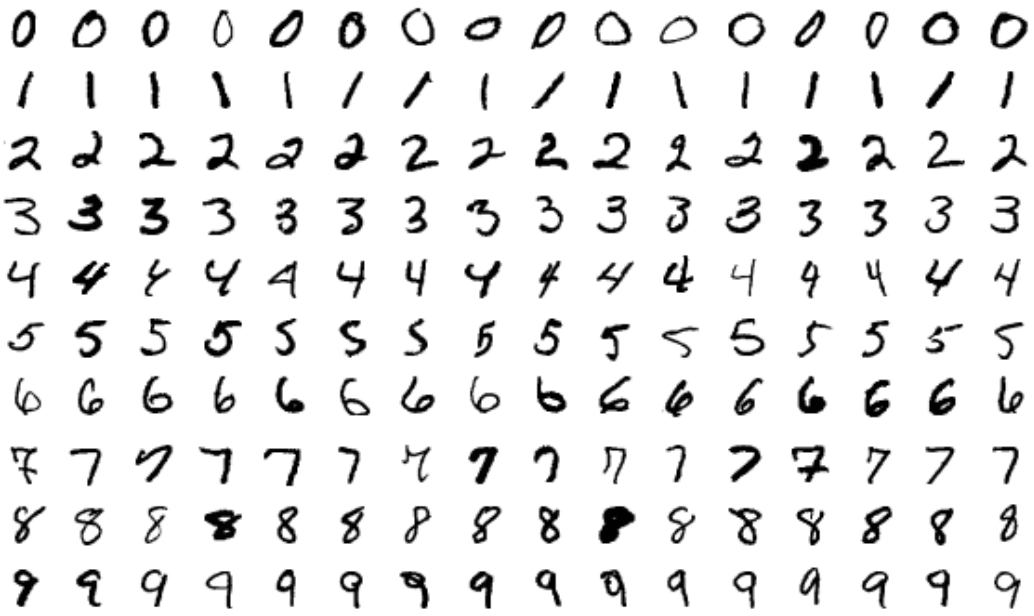
Pour limiter les problèmes liés au bruit (ce qu'on appelle ***Over fitting***, et que nous verrons dans le chapitre 7) on peut demander à l'algorithme de trouver les **K** voisins les plus proches du point vert.

Cela **améliore** la qualité du modèle car il devient moins sensible aux impuretés et cas particuliers qui viendraient empêcher la bonne **généralisation** (Chapitre 7).



Vision par ordinateur avec K-NN dans Jupyter

Cette fois ci, je vous propose de développer un programme capable de reconnaitre un chiffre entre 0 et 9 écrit à la main. Fini les données générées aléatoirement ! Voici les chiffres que la machine saura reconnaitre dans quelques minutes.



Vous pouvez charger ces données depuis Sklearn (la librairie contient des Datasets de base).

Commençons par importer les libraires habituelles :

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

Le code ci-dessous montre un exemple de chiffre présent dans le Dataset, c'est un exemple du chiffre 0.

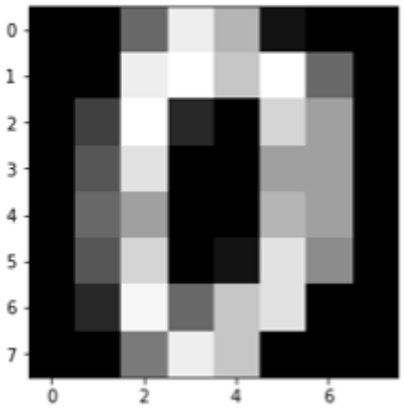
On apprend aussi que le Dataset comprend 1797 exemples, c'est-à-dire 1797 images, et que chaque exemple contient 64 *features*.

Que sont ces 64 *features* ? il s'agit de la valeur de chacun des 64 pixels qui forment les images.

Quand on soumet un nouveau chiffre à la machine, l'algorithme de 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.

```
# 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>
```



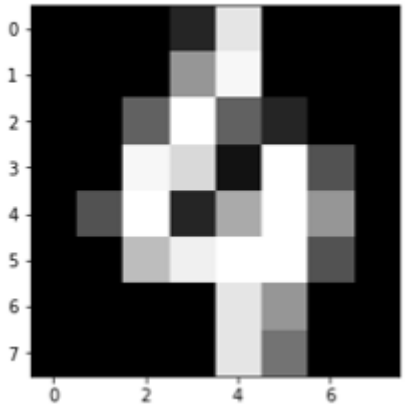
L'étape suivante consiste à entraîner le modèle de Nearest Neighbour. En exécutant le code vous-même, vous devriez obtenir un score de 99%, ce qui signifie que votre modèle reconnaitra le bon chiffre 99% du temps. Perso, je trouve ça impressionnant. Aujourd'hui, vous pourrez clairement dire que vous savez faire du Machine Learning !

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

# Entraînement du modele
model = KNeighborsClassifier()
model.fit(X, y)
model.score(X, y)
```

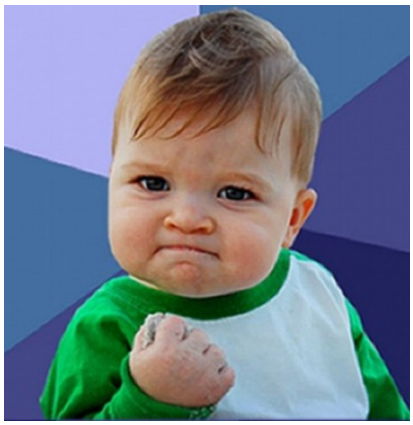
Pour finir en beauté, testons une image au hasard et voyons si la machine arrive à identifier de quel chiffre il s'agit. En l'occurrence, j'ai choisi de tester la 100ième image de notre Dataset, qui est un 4… et la machine a su la reconnaître !

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

```
array([4])
```



Bravo pour avoir su développer votre premier **vrai** programme de vision par ordinateur !



Bilan de ce chapitre

Dans ce chapitre, vous avez appris 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 Machine Learning. C'est elle que l'on trouve au cœur des neurones des fameux Réseaux de Neurones, dont nous allons parler dans le prochain chapitre.

Chapitre 5 : Réseaux de Neurones

Dans ce dernier chapitre sur l'apprentissage supervisé, nous allons démystifier les fameux **Réseaux de** N**eurones**. Ces modèles qui font le buzz aujourd'hui sont utilisés pour la reconnaissance vocale, la vision par ordinateur et autres applications complexes.

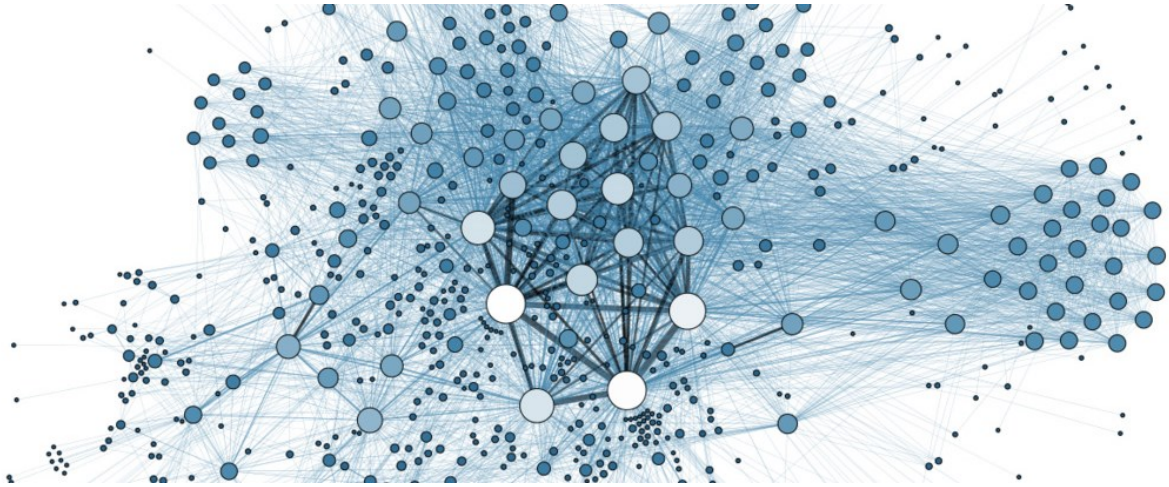
Vous allez apprendre :

- Ce qu'est le Deep Learning
- Ce qui compose un Réseau de Neurones dans les détails
- Comment programmer votre premier Réseau de Neurones

Introduction aux Réseaux de Neurones

Ah ! Il est temps de sortir l'artillerie lourde avec les **Réseaux** de **Neurones** (Neural Network) qui font aujourd'hui le succès du **Deep Learning**.

Les Réseaux de Neurones sont des modèles bien plus **complexes** que tous les autres modèles de Machine Learning dans le sens où ils représentent des fonctions mathématiques avec des **millions** de coefficients (les paramètres). Rappelez-vous, pour la régression linéaire nous n'avions que 2 coefficients $a \; et \; b$…



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

- La reconnaissance d'objets et reconnaissance 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 plusieurs jours)
- Une plus **grande** puissance de calcul.

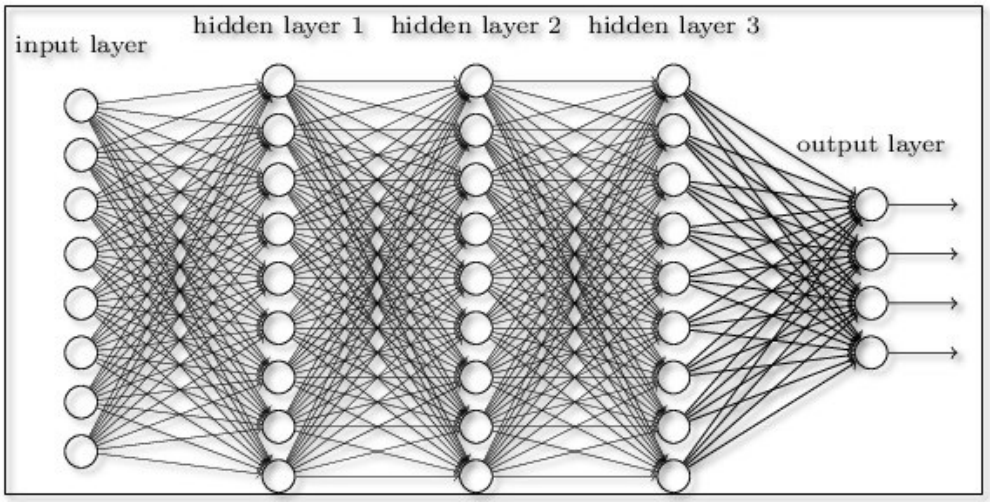
Pour dépasser ces challenges, les chercheurs dans le domaine ont développés des **variantes** du Gradient Descent ainsi que d'autres techniques pour calculer plus rapidement les dérivées sur des millions de données. Parmi ces solutions on trouve :

- **Mini**-**Batch Gradient Descent** : Technique pour laquelle le Dataset est fragmenté en petits lots pour simplifier le calcul du gradient à chaque itération.
- **Batch Normalization**: Mettre à la même échelle toutes les variables d'entrée et de sortie internes au Réseau de Neurone 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.

Sans plus tarder, voyons l'anatomie d'un Réseau de Neurones pour démystifier ce concept.

Comprendre les Réseaux de Neurones

Voilà à quoi ressemble un Réseau de Neurones :



Vous remarquez 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'activation**. Pour un réseau de neurone basique, la **fonction Logistique** est utilisée comme fonction d'activation. C'est pour cela que nous l'avons vue dans le chapitre 4.

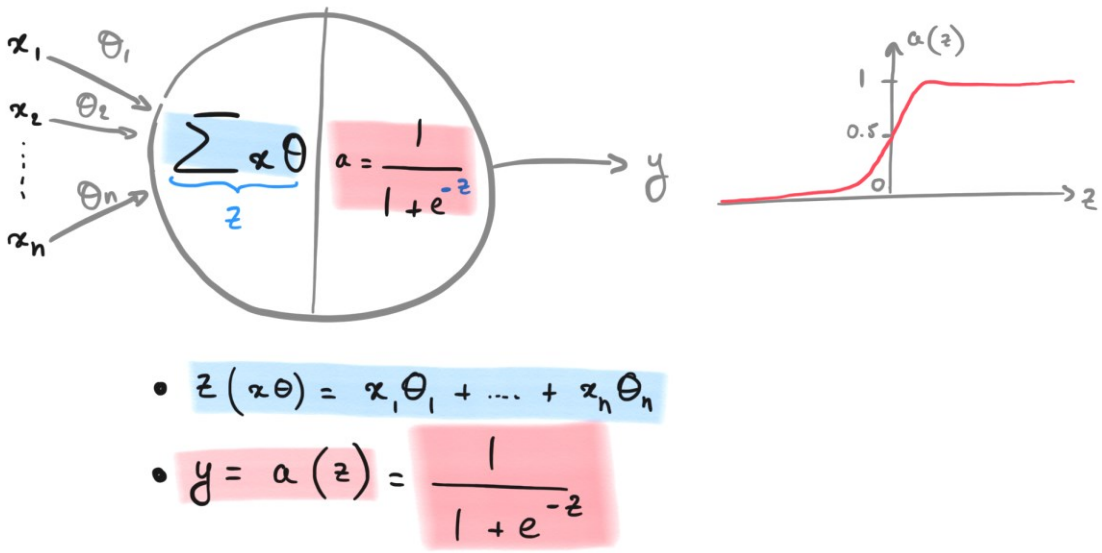
Commençons par analyser ce qui se passe dans **1 neurone**.

Réseau de Neurone à 1 Neurone : Le perceptron

Le réseau de Neurones le plus simple qui existe porte le nom de perceptron. Il est **identique** à la Régression Logistique du chapitre 4.

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

1. Le neurone calcule la $\boldsymbol{somme}\ \boldsymbol{z}$ de toutes les entrées $\boldsymbol{z} = \sum \boldsymbol{x\theta}$**.** 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 rapides** :

- La fonction tangente hyperbolique $\tanh(z)$
- La fonction $Relu(z)$

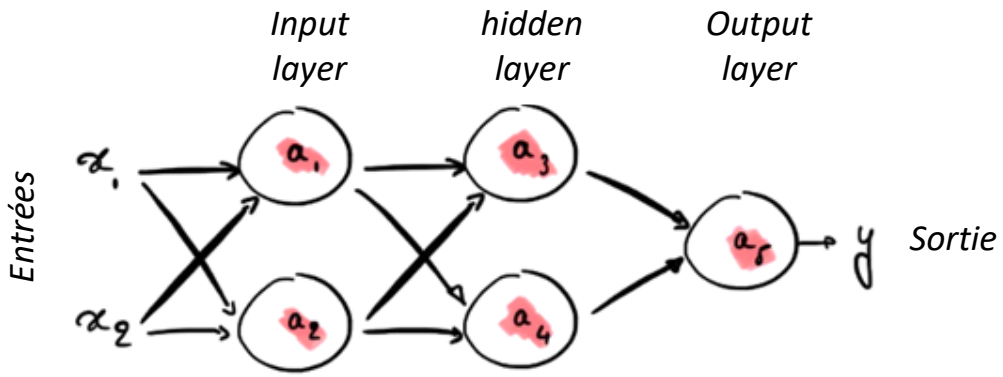
Réseaux à plusieurs neurones : le Deep Learning

Pour créer un Réseaux de Neurones, il suffit de développer plusieurs de ces perceptrons et de les **connecter** les uns aux autres d'une 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 de droite qui suit.

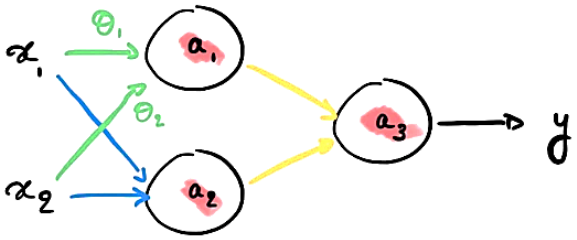
On peut ainsi construire un réseau avec autant de couches et de neurones que l'on veut. Plus il y a de couches, plus on dit que le réseau est **profond** (*deep*) et plus le modèle devient **riche**, mais aussi **difficile** à entraîner. C'est ça, le **Deep Learning**.

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 leur entrées/sorties, qui sont utilisées par les *layers* suivants.



Dans les détails, un réseau plus simple (à 3 neurones) nous donnerait la sortie $\boldsymbol{a_3 = \sigma(\theta_1 a_1 + \theta_2 a_2)}$ où $\boldsymbol{\theta_1}$ et $\boldsymbol{\theta_2}$ sont les coefficients liés aux connections entre neurones $\boldsymbol{a_1 \rightarrow a_3}$ et $\boldsymbol{a_2 \rightarrow a_3}$. Ce sont les **paramètres** de notre modèle.

Dans le réseau suivant, on a donc 6 paramètres (que je différencie par les couleurs, la réelle annotation des paramètres étant plus complexe).



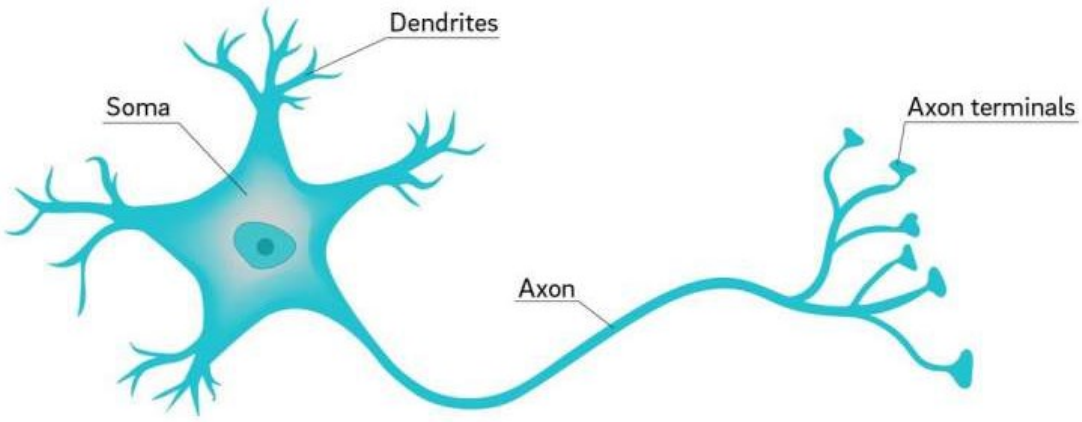
$$a_1 = \sigma\left(\theta_1 x_1 + \theta_2 x_2\right)$$
$$a_2 = \sigma\left(\theta_1 x_1 + \theta_2 x_2\right)$$
$$a_3 = \sigma\left(\theta_1 a_1 + \theta_2 a_2\right)$$

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 aujourd’hui présentée 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 neurone).

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



En réalité, les Réseaux de Neurones n’ont **rien à voir** avec le cerveau humain. Désolé…

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.

Pourquoi utilise-t-on alors l'expression de Réseau de Neurone ? Sûrement parce que l'analogie avec les neurones facilite la compréhension de ce type de modèle, mais également parce que l'utilisation de ce **Buzz word** a permis de solliciter l'intérêt des journalistes et des entreprises dans les années 2010. Une histoire de marketing…

L'entraînement d'un Réseau de Neurone

Rappelez-vous : Pour résoudre un problème de Supervised Learning, il vous 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 $(\boldsymbol{X}, \boldsymbol{y})$ comme pour les autres problèmes. Les features $(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \dots)$ sont distribuées à l'entrée du réseau (dans le premier *layer*)

- Pour programmer le **Modèle**, il faut emboîter les fonctions des différent niveaux d'activation les unes dans les autres comme je l'ai montré plus haut pour l'exemple des 3 neurones. C'est ce qu'on appelle ***Forward Propagation*** (faire le chemin des entrée $\boldsymbol{X}$ vers la sortie $\boldsymbol{y}$).

- Pour exprimer la Fonction Coût et son gradient, c'est mathématiquement délicat. Il faut calculer la **contribution** de chaque neurone dans **l'erreur** finale. Tout ce que vous avez à savoir, c'est que cela est possible avec une technique appelée ***Back Propagation*** (faire le chemin dans le sens inverse : $\boldsymbol{y}$ vers $\boldsymbol{X}$).

- Enfin, pour minimiser la Fonction Coût, il suffit d'utiliser Gradient Descent en utilisant les gradients calculés avec Back Propagation. Le Gradient Descent en lui-même n'est **pas différent** de celui que nous avons vu dans le chapitre 2 (bonne nouvelle).

Programmer votre premier Réseau de Neurones pour identifier des espèces d'Iris.

Normalement, on développe des Réseaux de Neurones avec un Framework comme **Tensorflow**, mais l'apprentissage de cet outil dépasse un peu le cadre de ce livre. Cependant, je veux bien me faire violence en vous montrant comment faire avec **Sklearn** ☺. Pour cela, il faudra importer **`MLPClassifier`** (qui signifie : *Multi-Layer Perceptron Classifier*).

Cette fois, je vous propose de développer un programme capable de reconnaitre une espèce parmi plusieurs de la famille des Iris. L'algorithme utilise 4 *features* pour effectuer son calcul :

- x_1 : La longueur du pétale
- x_2 : La largeur du pétale
- x_3 : La longueur du sépale
- x_4 : 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 donées
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*, j'utilise le code suivant : `hidden_layer_sizes=(10, 10, 10)`

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

Normalement, vous devriez obtenir un score de 98% pour votre Neural Network. Cela signifie que la machine arrive à prédire la bonne espèce de fleur 98% du temps.

Résumé de l'apprentissage supervisé

C'est avec les Réseaux de Neurones que s'achève ces 4 chapitres sur **l'apprentissage supervisé**, qui est la technique **la plus utilisée** en Machine Learning et en Deep Learning. Rappelez-vous qu'il existe 2 familles de problèmes dans l'apprentissage supervisé :

- Les Régressions
- Les Classifications

Pour résoudre ces problèmes, ne perdez jamais de vue les **4 étapes essentielles** pour développer votre modèle :

1. Dataset
2. Modèle
3. Fonction Coût
4. Algorithme de minimisation

Avec Sklearn, vous pouvez développer des modèles de Machine Learning simplement en utilisant les fonctions que nous avons vues et qui intègrent directement les étapes 3 et 4. Il ne vous reste alors qu'à :

1. Importer un Dataset
2. Choisir un modèle parmi ceux proposés par **sklearn** :
   - `SGDRegressor()`
   - `KNeighborsClassifier()`
   - `MLPClassifier()`
   - Etc…
3. Utiliser la fonction model.**fit** pour effectuer l'apprentissage

Il existe beaucoup d'algorithmes de Régression et de Classification, je n'ai pas parlé de *Support Vector Machine* ou bien de *Random Forest*, mais vous pouvez trouver plus d'information à ce sujet sur machinelearnia.com

Dans le prochaine chapitre, vous allez découvrir une nouvelle technique d'apprentissage, dont la méthodologie est bien différente de celle que nous avons vue jusqu'à présent : **L'apprentissage non-supervisé.**

Chapitre 6 : Apprentissage Non-Supervisé

Dans ce chapitre, vous allez découvrir la deuxième technique d'apprentissage utilisée en Machine Learning : **L'apprentissage non-supervisé**.

Vous allez apprendre l'algorithme le plus populaire dans cette technique : le ***K-Mean Clustering***, qui permet de segmenter des données clients dans le monde du marketing, ou bien de faciliter la recherche scientifique en associant ensemble des molécules/matériaux/phénomènes semblables.

Unsupervised Learning

Le problème de l'Apprentissage Supervisé

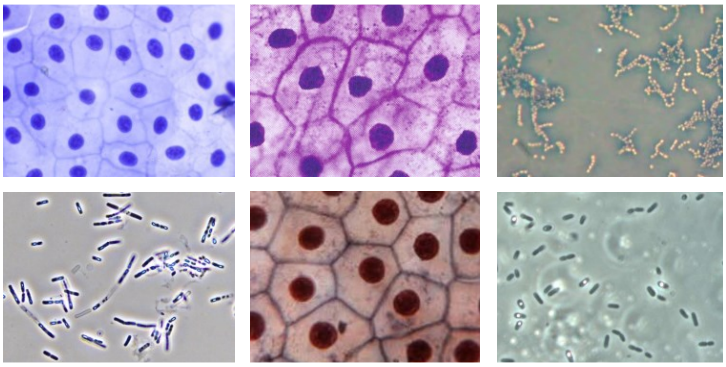
D'un certain point de vue, l'apprentissage supervisé consiste à enseigner à la machine des choses que nous **connaissons déjà**, étant donné que nous construisons à l'avance un Dataset qui contient des questions $\boldsymbol{X}$ et des réponses $\boldsymbol{y}$.

Que faire alors si vous disposez d'un Dataset sans valeur $\boldsymbol{y}$? Que faire si vous voulez que la machine vous aide à compléter vos connaissances en apprenant certaines choses que vous ignorez ?

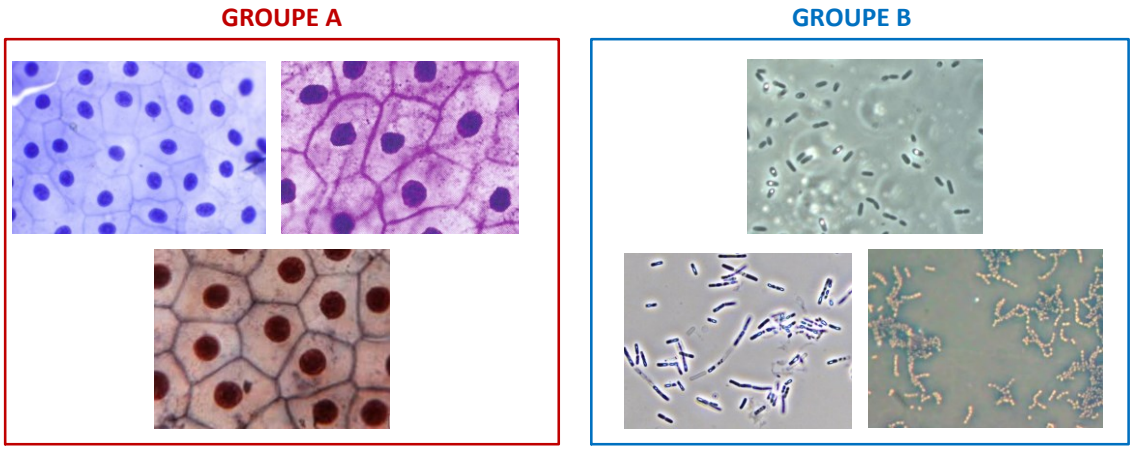
Rappelez-vous, dans le Chapitre 1, nous avons parlé d'apprendre la langue chinoise à partir d'un livre de traduction (x, y). Que faire alors si je vous retire le bouquin et que je vous envoie vivre seul en Chine ? Arriverez-vous à apprendre le chinois tout seul ? Il y a forcément un moyen d'y parvenir, c'est ce qu'on appelle l'apprentissage non-supervisé (**Unsupervised Learning**).

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

Regardez ces 6 photos. Pouvez-vous les regrouper en 2 familles selon leur ressemblance ?



Bien sûr ! C'est même plutôt simple. Nul besoin de savoir s'il s'agit de cellules animales, de bactéries ou de protéines pour apprendre à classer ces images**.** Votre cerveau a en fait reconnu des **structures communes** dans les données que vous lui avez montrées.



Dans l'apprentissage non-supervisé, on dispose ainsi d'un Dataset (x) sans valeur (y), et la machine **apprend** à reconnaitre des **structures** dans les données (x) qu'on lui montre.

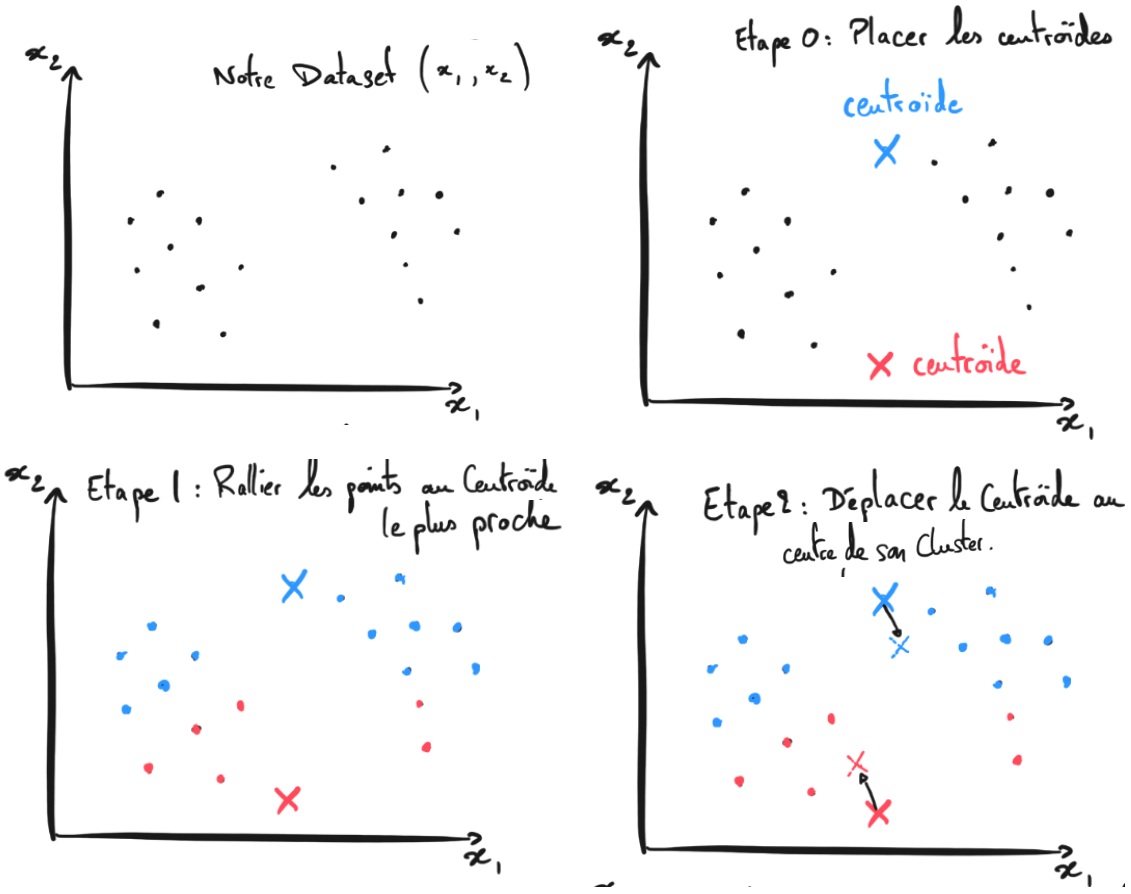
On peut ainsi regrouper des donnés dans des clusters (c'est le **Clustering**), détecter des **anomalies**, ou encore **réduire** la **dimension** de données très riches en compilant les dimensions ensembles.

Algorithme de K-Mean Clustering

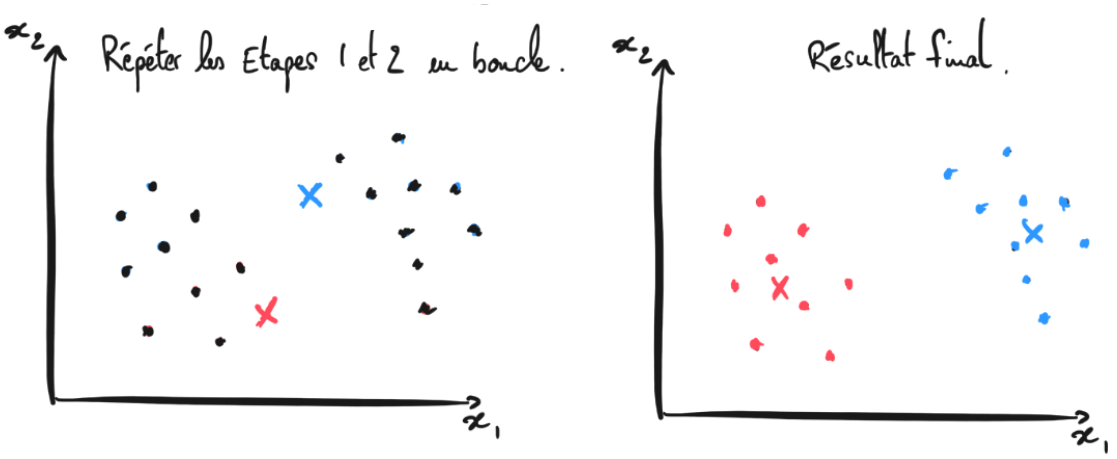
Le *K-Mean Clustering* est sans doute l'algorithme le plus populaire pour les problèmes de **Clustering** (regrouper des données selon leur structure commune). Il est souvent utilisé en marketing pour cibler des groupes de clients semblables pour certaines campagnes publicitaires.

L'algorithme fonctionne en 2 étapes répétées en boucle. On commence par placer au hasard un nombre **K** de **centres** dans notre nuage de points. Dans l'exemple ci-dessous, K=2. Ensuite :

> L'étape 1 consiste à rallier chaque exemple au centre le **plus proche**. Après cette étape, nous avons **K Clusters** (ici 2 clusters)
> L'étape 2 consiste à déplacer les centres **au milieu** de leur Cluster.



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



Programmer un K-Mean Clustering

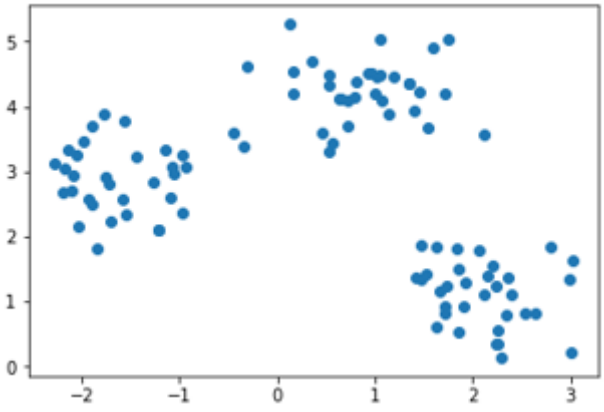
La librairie Sklearn nous permet aussi de faire du Unsupervised Learning !

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, nous allons créer un Dataset de 100 exemples à 2 features, en simulant 3 clusters.

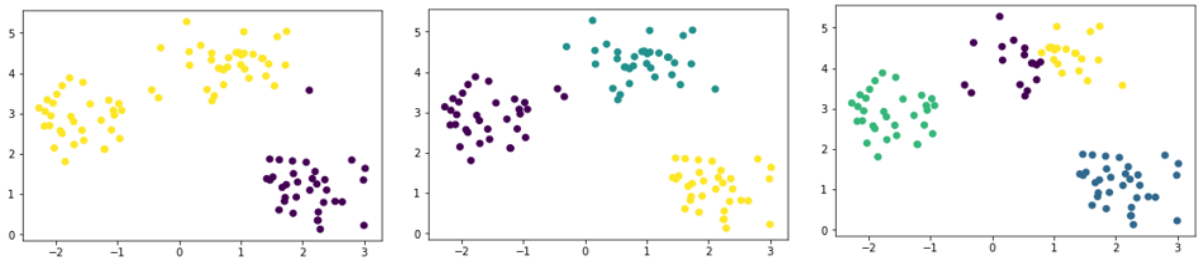
```
# Générer des données:
X, y = make_blobs(n_samples=100, centers = 3, cluster_std=0.5, random_state=0) #nb_feat
ures = 2 par défaut
plt.scatter(X[:,0], X[:, 1])
```



Dans le code ci-dessous, on entraîne un modèle de *K-Mean Clustering* à 3 centres (K=3). J'ai aussi affiché les résultats que l'on obtient avec K = 2 et K = 4.

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

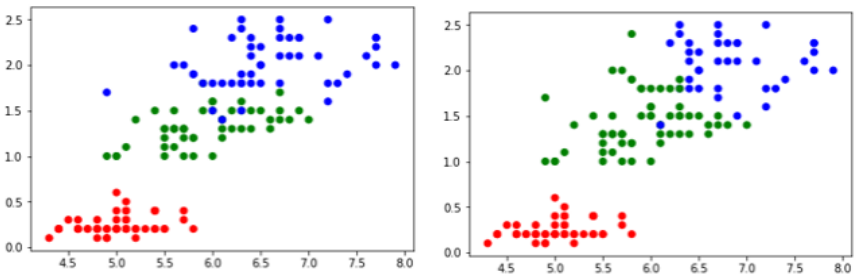
```
#Visualiser les Clusters
predictions = model.predict(X)
plt.scatter(X[:,0], X[:,1], c=predictions)
```



K = 2 **K = 3** **K = 4**

La prochaine fois que vous avez à disposition un tableau de données sur vos clients, sur les caractéristiques d'un produit, ou sur des documents que vous devez classer, pensez à utiliser le *K-Mean Clustering* pour laisser la machine proposer sa méthode de classement.

Pour finir, prenons le Dataset des fleurs d'Iris que vous avez su 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**

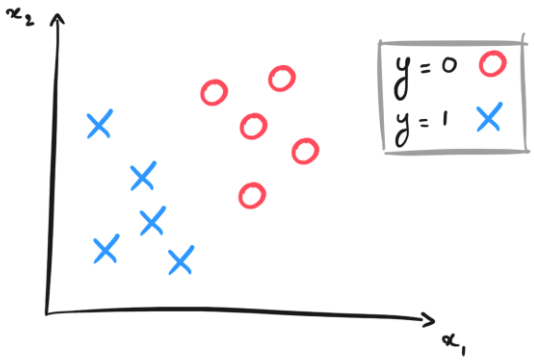
Les résultats sont très ressemblants, preuve qu'il est possible de faire de bonnes classifications même avec des Datasets (x) sans valeur (y) en utilisant l'apprentissage **non-supervisé**.

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*.

Apprentissage Supervisé
Labelled Dataset (x, y)



y	x_1	x_2
0	6	3
0	3	5
0	8	3
1	2	2
1	3	5
0	5	3

Apprentissage Non-Supervisé
Unlabelled Dataset (x)

x_1	x_2
6	3
3	5
8	3
2	2
3	5
5	3

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 (dont nous n'avons pas parlé). 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.

Maintenant que vous connaissez les principaux outils du Machine Learning, il est temps d'apprendre **comment se servir** de ces outils pour résoudre de vrais problèmes. C'est l'objet du chapitre final de ce livre !

Chapitre 7 : Comment gérer un projet de Machine Learning

Votre voyage arrive à sa fin. Au cours de ce livre, vous avez appris des algorithmes puissants et réellement utilisés dans l'industrie pour construire des modèles à partir de données. Mais un algorithme seul ne résout aucun problème. Il est temps d'apprendre **comment utiliser** vos connaissances pour **résoudre des problèmes**. Dans ce chapitre, vous allez apprendre :

- Comment préparer votre Dataset
- Comment lutter contre Le phénomène ***d'Over fitting***
- Comment diagnostiquer un modèle de Machine Learning avec le ***Train set*** et ***Test Set***
- Le cycle de développement d'un modèle de Machine Learning

L'erreur que font la majorité des Novices

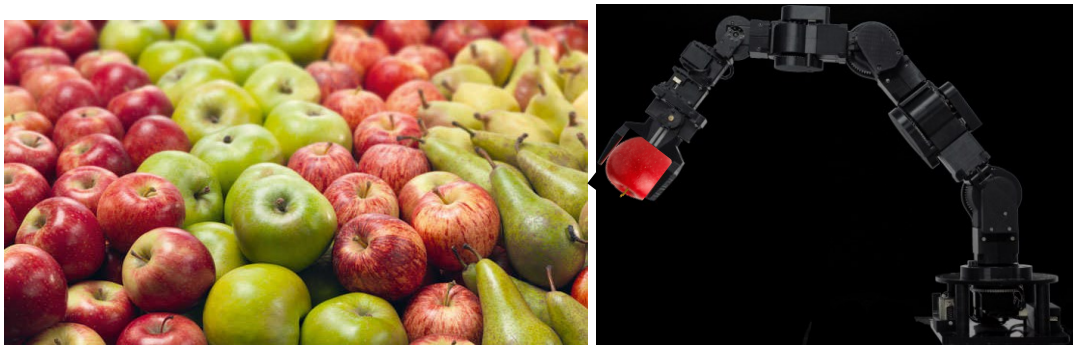
Beaucoup de gens travaillent longtemps leur maitrise des algorithmes au détriment de leur compétence à résoudre des problèmes.

Pour mener à bien un projet de Machine Learning, il ne suffit pas de connaitre des algorithmes (comme ceux que nous avons appris dans ce livre) mais il faut également savoir comment se servir de ces algorithmes, ce que **peu** de gens savent bien faire !

C'est ce que vous allez apprendre dans ce dernier chapitre, qui est l'un des chapitres les plus importants de ce livre.

Un exemple de scénario typique

Votre patron souhaite automatiser la logistique de son usine. Il vous confie un Dataset et vous demande de développer un modèle de Machine Learning pouvant identifier différents fruits (pommes, poires, etc.) pour que la machine puisse les ranger automatiquement dans la bonne boîte.



Vous entraînez un modèle complexe (par exemple un réseau de neurones avec beaucoup de *layers*) suffisamment longtemps pour minimiser la Fonction Coût et vous obtenez une précision de **99%**. Bien joué !

Vous livrez ce modèle à votre patron, mais quelques jours plus tard il revient très mécontent : votre modèle n'est pas aussi bon que vous le prétendez, il a une précision de **70%**.

Que s'est-il passé ?

Je vais vous dévoiler les causes plausibles et les erreurs à ne pas commettre pour éviter ce genre de situation, puis les solutions à ces challenges. 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

"It's not who has the best algorithm that wins. It's who has the most data."
Andrew Ng

Une étude menée en 2001 par Michelle Banko et Eric Brill montre que la performance d'un programme de Machine Learning dépend avant tout de la **quantité** de données que comporte votre Dataset. Cela explique en partie l'obsession qu'on les géants du Web (les GAFAM) à récolter des quantités colossales de données.

L'étude révèle aussi que beaucoup d'algorithmes de Machine Learning sont **similaires** en terme de performance.

Conséquence : Si vous avez plus de données que votre concurrent, vous êtes **vainqueur**, même si votre concurrent à un meilleur algorithme !

Mais avoir beaucoup de données ne suffit pas, il faut aussi avoir de **bonnes** données et **comprendre** ces données. C'est ce point qui fait défaut à de nombreux Data Scientists. Pourtant, l'étape de préparation des données (*Data pre-processing*) représente le plus grand temps passé sur un projet de Machine Learning, voyons pourquoi.

Data pre-processing: Comment préparer votre Dataset

Lorsque vous recevez un Dataset, il est impératif de procéder à quelques retouches **avant** de commencer à faire du Machine Learning. Voici une liste des actions à compléter pour bien préparer son Dataset :

- Il est fréquent qu'un Dataset contienne quelques **anomalies**, voire des erreurs, qu'il faut **supprimer** pour ne pas biaiser l'apprentissage de la machine (vous ne voudriez pas que la machine apprenne quelque chose de faux).

- Il est aussi important de **normaliser** vos données, c'est-à-dire les mettre sur une même échelle pour rendre l'apprentissage de la machine plus rapide et aussi plus efficace.

- Si vous avez des valeurs **manquantes**, il faut être capable de leur assigner une valeur défaut.

- Si vous avez des *features* **catégoriales** (exemple : homme/femme) il faut les convertir en données numériques (homme=0, femme=1).

- Egalement, il est très important de nettoyer le Dataset des *features* **redondantes** (qui ont une forte corrélation) pour faciliter l'apprentissage de la machine.

- Finalement, un point qui peut faire toute la différence est la création de nouvelles features, ce qu'on appelle ***feature engineering***.
  Exemple : Prenez un Dataset immobilier qui contient les *features :*
  - $x_1 = longueur\ jardin$
  - $x_2 = largeur\ jardin$

  Alors il est possible de créer $x_3 = x_1 \times x_2$ 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* dans Jupyter, utiliser la librairie **pandas**.

```
import pandas as pd

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

Attention ! Le tableau que vous cherchez à importer doit convenir à un certain format. Pour vos débuts, assurez-vous d'avoir un fichier Excel qui ne contient que vos données (pas de notes ni commentaires) et qui commence dès la colonne A ligne 1 :

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
18							

VS

Exemple à NE PAS suivre

	A	B	C	D	E	F	G	H	I
1									
2									
3		Appartement Région Paris							
4									
5		prix	surface	qualite	annee	jardin	garage	distance parc	
6		313000	124	3	1.5	0	oui	150	
7		2384000	339	5	2.5	70	oui	1300	
8		342000	179	3	2	0	oui	30	
9		420000	186	3	2.25	30	oui	2175	
10		550000	180	4	2.5	25	oui	400	
11		490000	82	2	1	10	non	380	
12		335000	125	2	2	0	non	160	dégat des eaux!
13		482000	252	4	2.5	0	oui	210	
14		452500	226	3	2.5	32	oui	230	
15		640000	141	4	2	0	non	25	
19		365000	101	3	1	0	non	96	
20		1200000	270	5	2.75	50	oui	145	
21		242500	111	3	1.5	5	non	1450	
22									
23				Penser a appeller Michel pour le dernier appartement !!!					
24									

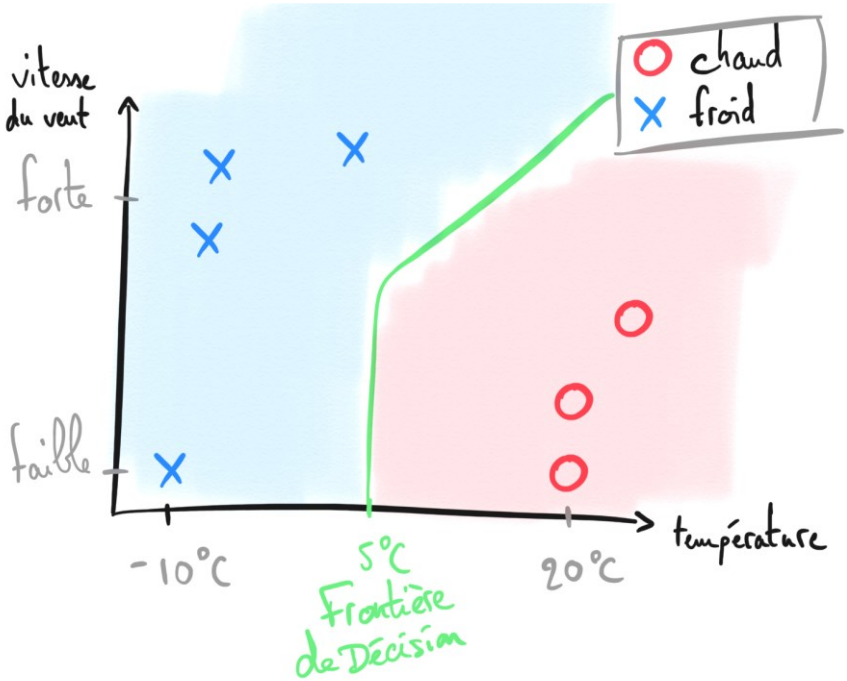
Vous pouvez ensuite utiliser certaines fonctions pour **nettoyer** votre Dataset, convertir les **catégories** en valeurs numériques, et charger votre Dataset dans une ***matrice X*** et un ***vecteur y*** pour commencer le Machine Learning !

```
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 Machine Learning, 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, vous avez une certaine expertise qui vous permet de rejeter la réponse de la machine, et vous comprenez que le Dataset a besoin d'être complété avec des données supplémentaires.

Etes-vous certain d'avoir ce niveau d'expertise en médecine pour développer un modèle qui pourrait décider de la vie d'un patient ?

Conclusion : vous serez un meilleur Data Scientist en travaillant sur les projets qui se rattachent à vos **compétences techniques**.

Les données doivent toujours venir de la même distribution

Une dernière erreur que je veux partager avec vous et qui peut causer une chute de performance assez importante est l'utilisation d'un modèle de machine Learning sur des données provenant d'une **autre distribution** que les données sur lesquelles la machine a été entraînée.

Par exemple, si vous développez un modèle pour reconnaitre 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 reconnaitre les *features* qu'il a appris durant l'entrainement 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.

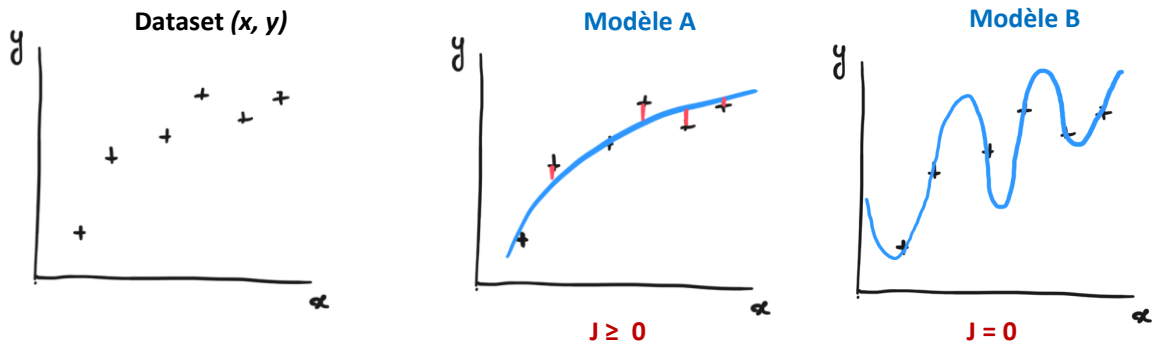
Le travail de préparation des données prend en général **80% du temps de travail** d'un Data Scientist, mais s'il est bien fait, alors vous n'avez plus aucun problème par la suite.

Plus aucun problème… sauf un : L'***Over fitting***. J'ai voulu garder ce point-là pour le sprint final du livre ! Vous êtes prêts ?

Over fitting et Régularisation

Dans les chapitres d'apprentissage supervisé, nous avons développé des modèles en cherchant à **minimiser** les erreurs avec le Dataset. J'ai même affirmé (page 15) qu'un bon modèle nous donne de petites erreurs. En fait, ce n'est pas si simple…

Que pensez-vous des deux modèles ci-dessous ?



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

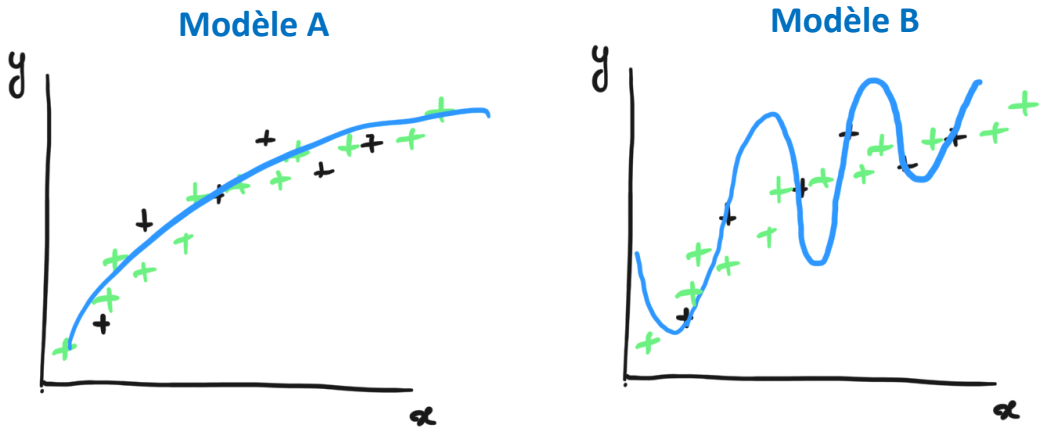
Pourtant, le modèle ***A*** semble plus convaincant, alors que celui-ci donne une Fonction Coût **plus élevée**.

Le modèle ***B*** souffre ici d'un problème appelé ***Over fitting***, qui est un phénomène **très courant** en Machine Learning et qu'il vous faut **absolument éviter**.

Over fitting : A vouloir aller trop loin, on va TROP loin.

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.

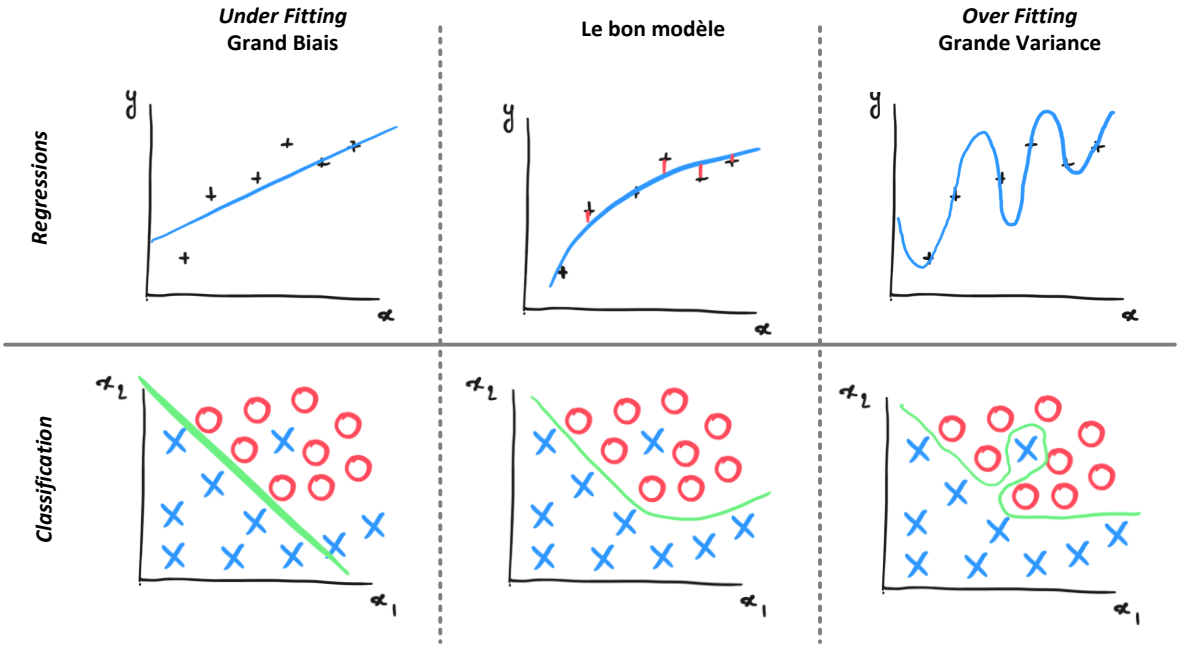


Sur le Dataset B était meilleur que A.
Mais sur de nouvelles données, A est meilleur que B

On pourrait alors se dire qu'il suffit de développer des modèles moins complexes avec moins de *features…* Et Pouf ! Plus de problème de variance !

C'est vrai, Mais on risque alors d'avoir un modèle erroné qui manque de précision. On appelle ça ***Under fitting***, 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 $\boldsymbol{\theta}$. Cette méthode porte justement le nom de **Régularisation**.

La Régularisation

La régularisation 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 :

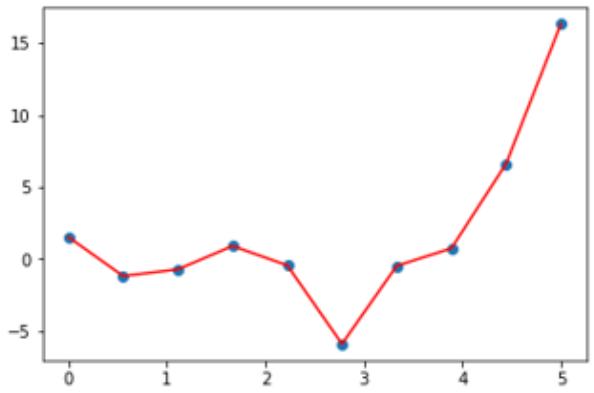
$$\boldsymbol{J(\theta) = \frac{1}{2m} \sum (F(X) - Y)^2 + \lambda \sum \theta^2} \ (Ridge\ ou\ L2\ Régularisation)$$

   Le facteur de régularisation $\boldsymbol{\lambda}$ correspond au niveau de pénalité : s'il est trop grand, on risque *l'Under fitting*, et 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 vous donnera en développant un modèle polynômial de degré 10 sur le Dataset suivant :

```
Coeff R2 = 1.0

[<matplotlib.lines.Line2D at 0x2300c670ef0>]
```

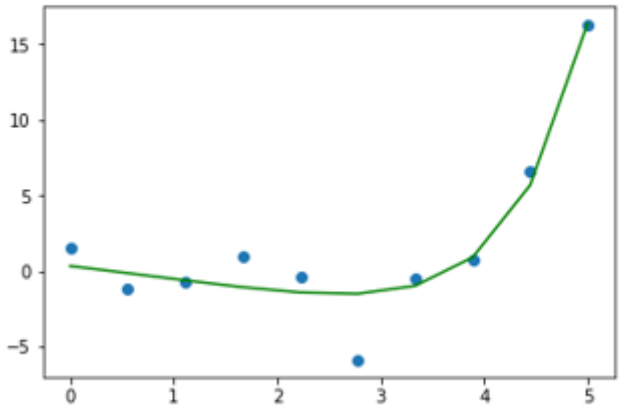


```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# 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


# Entraînement du modele. Ici on utilise les Equations Normales (LinearRegression)
# Les Equations normales reposent sur la méthode des moindres carrées, 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, vous pouvez 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>]
```



Vous savez maintenant que la performance **réelle** d'un modèle de Machine Learning ne repose pas simplement sur sa Fonction Coût : vous risquez *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 votre modèle aura sur des données **futures**, c'est-à-dire des données sur lesquelles il n'aura **pas été entraîné** ?

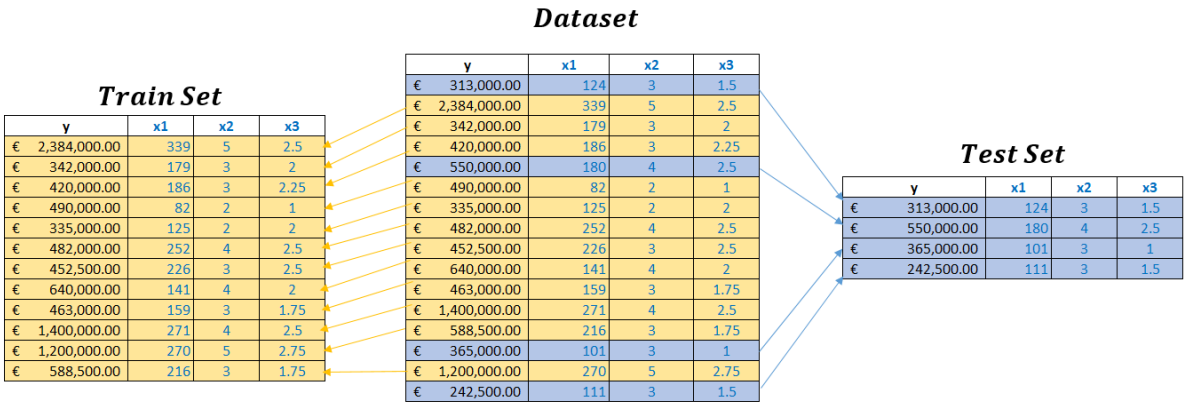
La réponse est dans la question ! Il faut entraîner votre modèle sur **une partie seulement** du Dataset et utiliser la seconde partie pour évaluer la vraie performance de notre modèle. On appelle cela le *Train set* et le *Test set.*

Diagnostiquer un modèle de Machine Learning

Train set et Test set

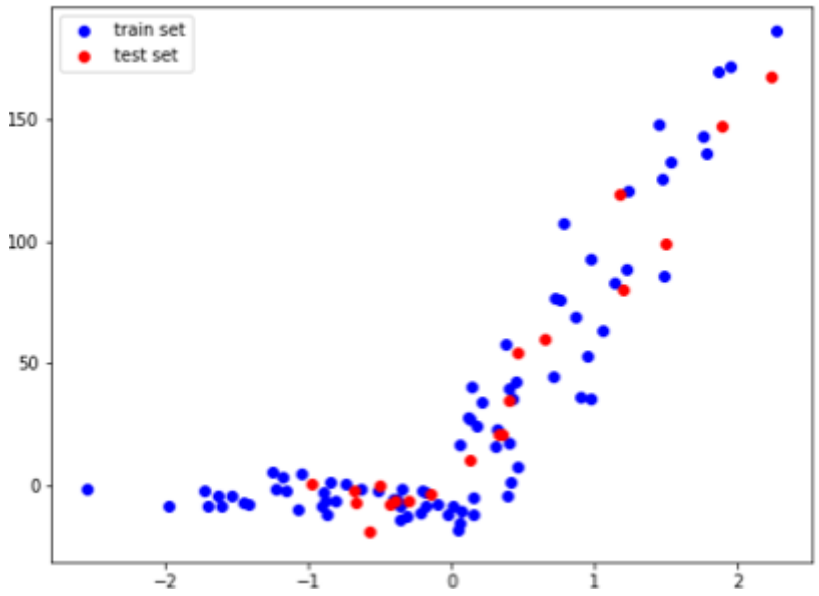
La bonne manière de mesurer la performance de votre modèle de Machine Learning est de tester celui-ci sur des données qui n'ont pas servi à l'entraînement. On divise ainsi le Dataset **aléatoirement** en deux parties avec un rapport 80/20 :

- **Train set (80%)**, qui permet à la machine d'entraîner un modèle.
- **Test set (20%)**, qui permet d'évaluer la performance du modèle.



Pour créer un *Train set* 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()
```

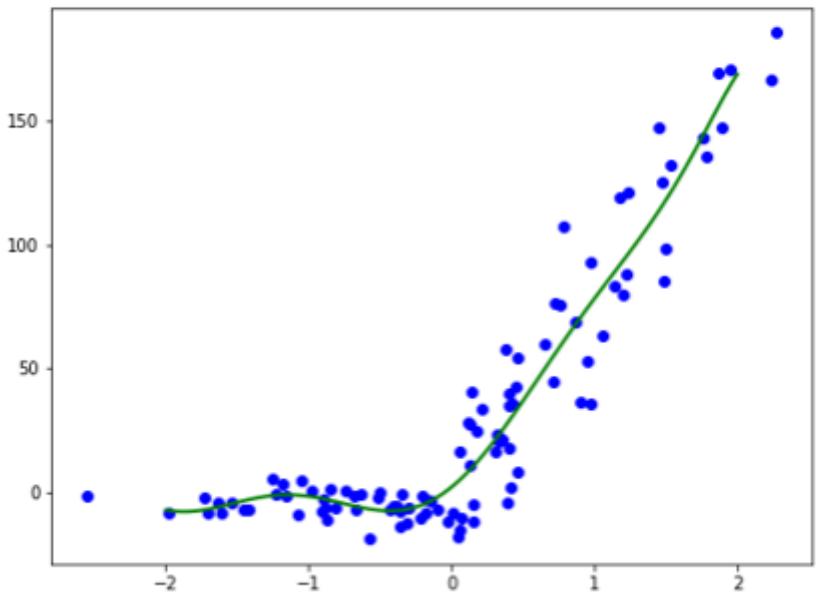


On peut ensuite entraîner notre modèle sur le $\boldsymbol{Train\ Set}$: (x_{train}, y_{train}), puis l'évaluer sur le $\boldsymbol{Test\ Set}$: (x_{test}, y_{test}) .

Dans le code ci-dessous je décide de mettre en pratique tout ce que nous avons vu : Je crée plusieurs *features* polynômiales (degré 10) et j'utilise la régularisation de Ridge pour éviter *l'Over fitting*. On obtient un score de **92%** pour l'entraînement et un score de **91%** pour l'évaluation sur des données nouvelles.

```
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)
```

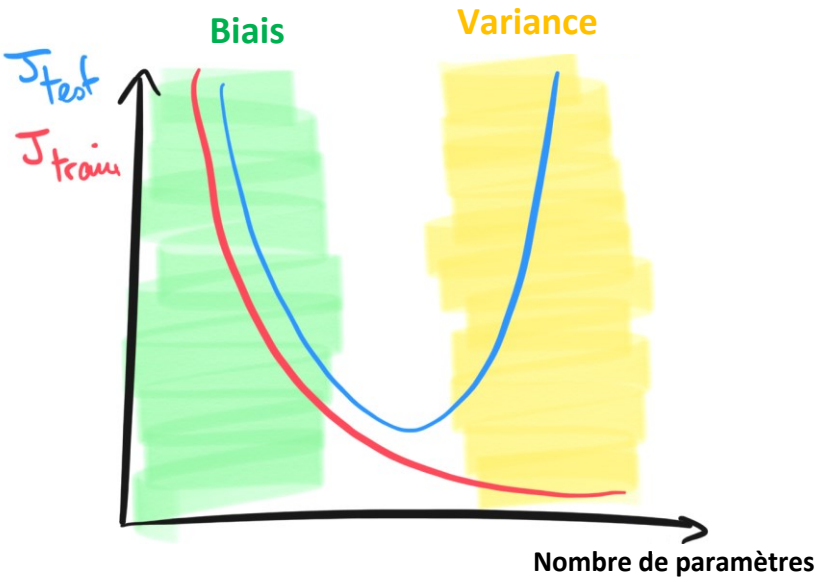
```
Coefficient R2 sur Train set: 0.9235739839329026
Coefficient R2 sur Test set: 0.9142326950179189

[<matplotlib.lines.Line2D at 0x230151d3080>]
```



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* $J(\theta)_{train}$ et le *Test set* $J(\theta)_{test}$:

- 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 *Under fitting* ?

Dans le cas où votre modèle a un grand biais (*Under fitting*) vous pouvez :

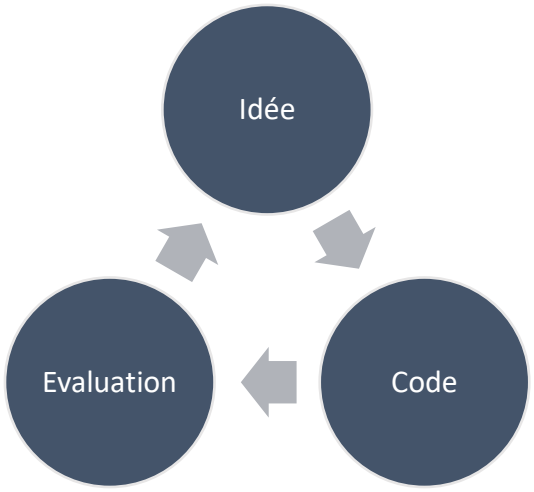
- Créer un modèle plus complexe, avec plus de paramètres.
- Créer plus de *features* à partir des *features* existantes.
- Entraîner votre modèle plus longtemps.
- Diminuer le *Learning Rate* du *Gradient Descent* (si le *Learning Rate* est trop grand, la Fonction Coût ne converge pas)
- Récolter plus de *features* dans les données (parfois une *feature* importante n'a pas été récoltée)

A l'inverse, si votre modèle a une grande variance (*Over fitting*) vous pouvez :

- 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 Machine Learning 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.

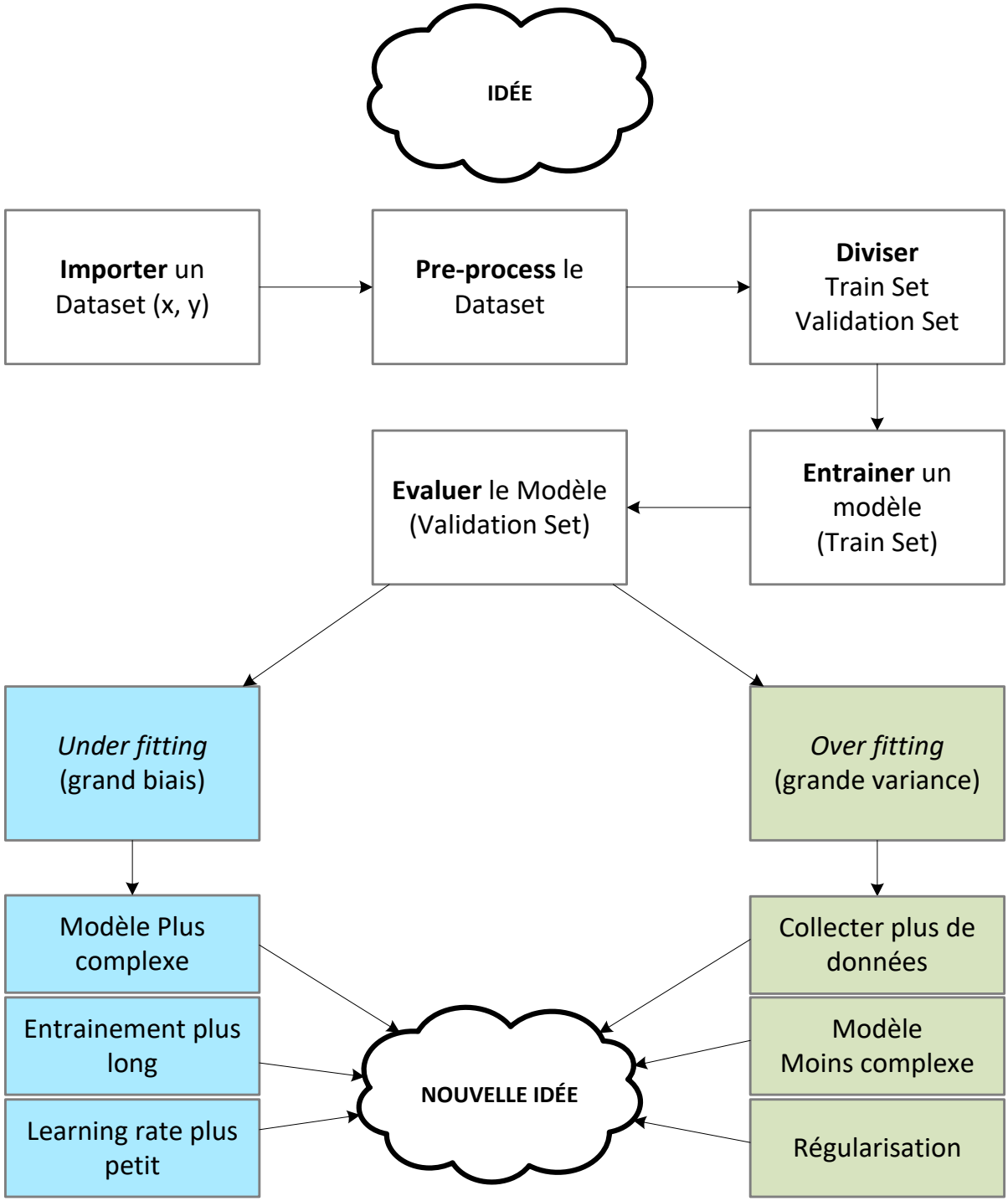


Le diagramme de la page suivante résume le cycle de développement du Machine Learning avec toutes les étapes que vous avez apprises, et marque la fin de ce chapitre final.

Notre voyage s'achève ici, mais j'espère que vous continuerez à tracer votre chemin dans le Machine Learning après avoir lu ce livre, et me soutenir sur machinelearnia.com et YouTube est le meilleur moyen de me remercier pour ce livre gratuit.

N'omettez pas de lire la conclusion du livre, je garde pour vous un petit **bonus !**

Résumé des étapes de développement en Machine Learning



CONCLUSION

Bravo ! Vous avez lu ce livre en entier, ce qui vous distingue de la majorité des gens qui commencent un livre sans jamais le finir. Le simple fait de l'avoir terminé doit vous conforter dans l'idée que vous avez les capacités de devenir un **excellent Data Scientist** !

Le Monde, et tout particulièrement la France, **manque cruellement de Data Scientists** ! Il y a énormément de places à prendre sur le marché du travail, mais d'ici 5 ans cela aura certainement changé.

Agissez donc MAINTENANT. Utilisez les techniques que vous avez apprises pour des tâches au travail et continuer d'apprendre grâce à mon site Internet : **machinelearnia.com**. Pour finir, je vous livre quelques conseils et un formulaire de Machine Learning.

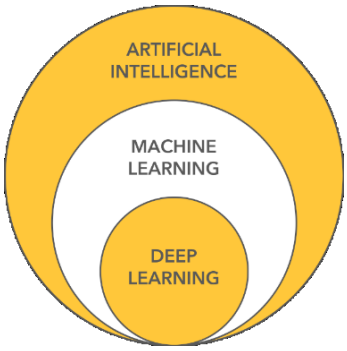
Du fond du cœur, **MERCI.**

Guillaume Saint-Cirgue

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 **Machine Learning**, très populaire depuis 2010. Le **Deep Learning** est un domaine du Machine Learning qui est focalisé sur le développement des réseaux de neurones et qui fait face à d'autres défis que ceux du machine Learning. Parmi ces défis, comment entrainer 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 Machine Learning on créé un programme qui acquiert une aptitude : conduire une voiture, voir des objets, etc.

Dataset :

Tableau de données $(\boldsymbol{X}, \boldsymbol{y})$ qui contient 2 types de variables :

Target $\boldsymbol{y}$
Features $\boldsymbol{X}$

On note $\boldsymbol{m}$ le nombre d'exemples que contient le tableau (le nombre de lignes) et $\boldsymbol{n}$ le nombre de *features* (le nombre de colonnes X).

Ainsi :

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

Pour désigner la *feature* $\boldsymbol{j}$ de *l'exemple* $\boldsymbol{i}$ on écrit $\boldsymbol{x}_{\boldsymbol{j}}^{(\boldsymbol{i})}$

Modèle :

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

On note $\boldsymbol{\theta}$ le **vecteur** qui contient les **paramètres** de notre modèle. Pour une régression linéaire, la formulation matricielle de notre modèle devient : $\boldsymbol{F(X) = X.\theta}$

Fonction Coût :

La Fonction Coût $\boldsymbol{J(\theta)}$ mesure l'ensemble des erreurs entre le modèle et le Dataset. De nombreux **métriques** d'évaluations peuvent être utilisés pour la Fonction Coût :

Mean Absolute Error (MAE)
Mean Squared Error (MSE) : Utilisée dans le Chapitre 3
Root Mean Squared Error (RMSE)
Accuracy : pour les classifications
Precision
Recall
F1 score.

Gradient de la Fonction Coût $\boldsymbol{\frac{\partial J(\theta)}{\partial \theta}}$

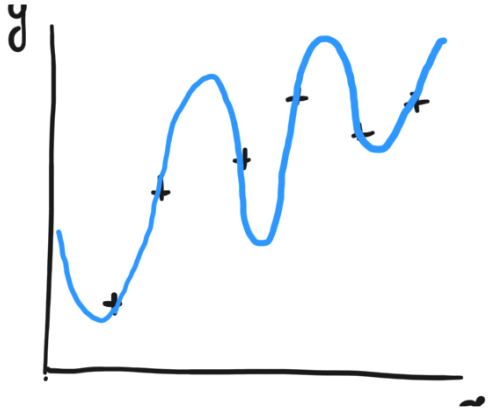
Gradient Descent :

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

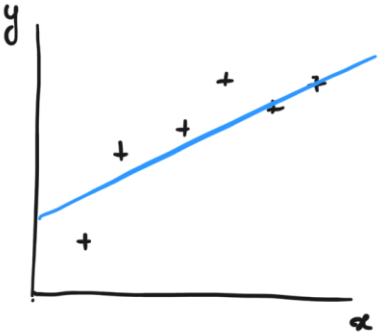
Variance :

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

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

$\boldsymbol{Modèle}: F(X) = X.\theta$

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

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

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

Résumé de la Régression Logistique

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

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

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

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

Code pour visualiser les courbes d'apprentissage sur une régression linéaire (Chapitre 3)

Copier/coller le code suivant dans Jupyter ou bien Spyder pour visualiser les courbes d'apprentissage.

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 $\boldsymbol{\theta}$ le vecteur qui contient les paramètres $\boldsymbol{a}$ et $\boldsymbol{b}$.

$$\boldsymbol{F = X.\theta}$$

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

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

```
#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

#definit 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 :

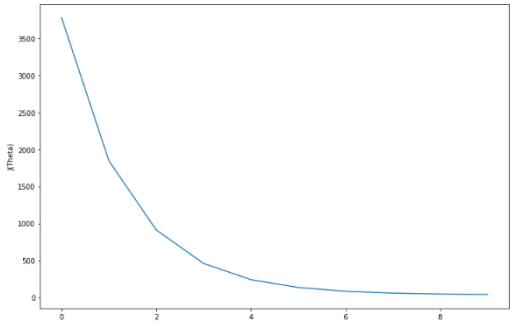
$$\theta = \theta - \propto Grad(\theta)$$

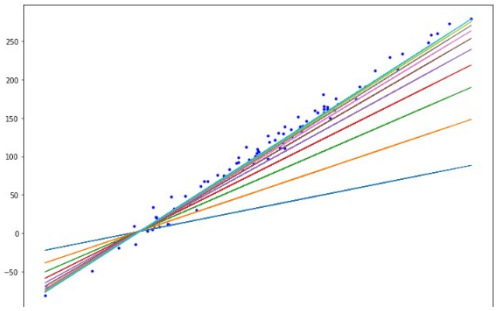
```
#algorithme de Gradient Descent
def gradient_descent(X, y, theta, learning_rate =0.001, iterations = 1000):
    m = len(y)
    cost_history = np.zeros(iterations)
    theta_history = np.zeros((iterations, 2))

    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=learni
ng_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 modele 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 Machine Learning:

Pour votre culture, voici d'autres algorithme de Machine Learning très populaires. Vous pouvez les apprendre en détail sur machinelearnia.com.

Support Vector Machine *:* Consiste à trouver la frontière de décision linéaire qui éloigne le plus les classes l'unes de l'autres. 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 votre Dataset en créant des variables non-corrélées.

Mes 3 conseils perso pour le Machine Learning

Conseil #1 : Notez toujours les dimensions de votre problème

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

A un moment ou un autre, vous aurez un *bug* parce que vous tentez de multiplier deux 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 vos 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

Vous connaissez la Loi de Pareto ? **80%** des effets sont le produit de **20%** des causes.

J'ai pu observer que ce principe s'applique très souvent en Machine Learning : 80% des erreurs de votre 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 votre programme est atteinte après seulement 20% de travail.

Faites alors le diagnostic de votre système en cherchant les points 80/20 Si vous voulez vous améliorer de 80% en produisant seulement 20% d'efforts. Bonne chance !

Conseil #3 : Philosophez et ne soyez pas obsédés par la performance

En Machine Learning, 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. Je vous invite à philosopher, à garder l'esprit ouvert et à passer plus de temps à chercher des solutions à d'autres problèmes.

Bonne chance à vous !