

Apprendre le
**MACHINE
LEARNING**
en une semaine



Guillaume Saint-Cirgue

Copyright © 2024 MACHINE LEARNIA

PUBLISHED BY GUILLAUME SAINT-CIRGUE

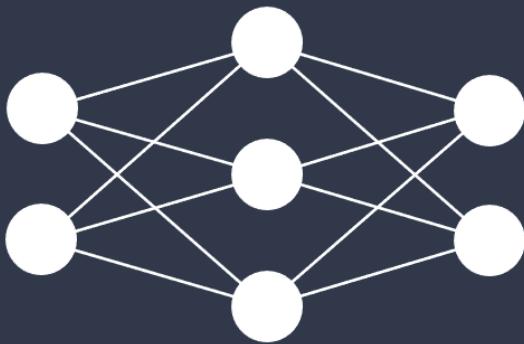
<https://www.machinelearnia.com>

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Table des matières

Table des matières	3
Avant de commencer	5
Les fondations du machine learning	8
Régression	22
Classification	37
Clustering	48
Détection d'anomalies	59
Apprentissage par renforcement	69
Deep learning et generative AI	81
Conclusion	91

Introduction



Avant de commencer

Je m'appelle Guillaume Saint-Cirgue et je suis Data Scientist au Royaume-Uni depuis 2016. Mais laissons de côté les présentations conventionnelles, car ceci n'est pas un CV 😊

Le machine learning n'est pas que mon métier. C'est ma vie. C'est mon ADN.

C'est pourquoi en 2019 j'ai créé la chaîne YouTube **Machine Learnia**, afin de partager ma passion de la data avec le reste du monde. Peu importe que mon contenu fasse des vues ou non, je voulais juste parler de ce qui m'anime au plus profond de moi.



Cinq ans plus tard, nous sommes plus de 150,000 abonnés dans la communauté!

Donc, inutile de me le cacher, je sais aujourd'hui que vous êtes, vous aussi, des **passionnés** du machine learning. Vous et moi, nous sommes pareils. 😊

C'est pourquoi j'ai décidé de faire plaisir à toute la communauté, ainsi qu'aux futurs arrivants, en écrivant une nouvelle version de mon livre *Apprendre le machine learning en une semaine*.

Durant ces dernières années, vous avez été des milliers à m'écrire pour me dire que ce livre avait été un véritable **déclencheur**. Aujourd'hui, beaucoup d'entre vous sont devenus data scientists confirmés. Certains ont même fondé leur entreprise. Quel parcours! Je suis fier de vous!

J'ai donc voulu revisiter le livre d'origine en y ajoutant de nouvelles illustrations, des codes prêts à l'emploi, ainsi que **trois nouveaux chapitres** exclusifs consacrés à la détection d'anomalies, à l'apprentissage par renforcement, et au Generative AI.

J'espère donc que vous prendrez autant de plaisir à lire ce livre, que j'en ai eu à l'écrire et à l'illustrer.

C'est parti pour **Apprendre** (ou ré-apprendre) **le Machine Learning en Une Semaine!** 😊

(Une note sur comment lire ce livre)

Tout comme dans la version originale, cette nouvelle version est composée de sept chapitres, chacun d'une dizaine de pages. Je vous encourage donc à lire un chapitre par jour afin de relever le défi : "Apprendre le machine learning en une semaine".

- Jour 1 : Les fondations du machine learning
- Jour 2 : La régression
- Jour 3 : La classification
- Jour 4 : Le clustering
- Jour 5 : La détection d'anomalies
- Jour 6 : L'apprentissage par renforcement
- Jour 7 : Le deep learning et gen AI

Les fondations du machine learning



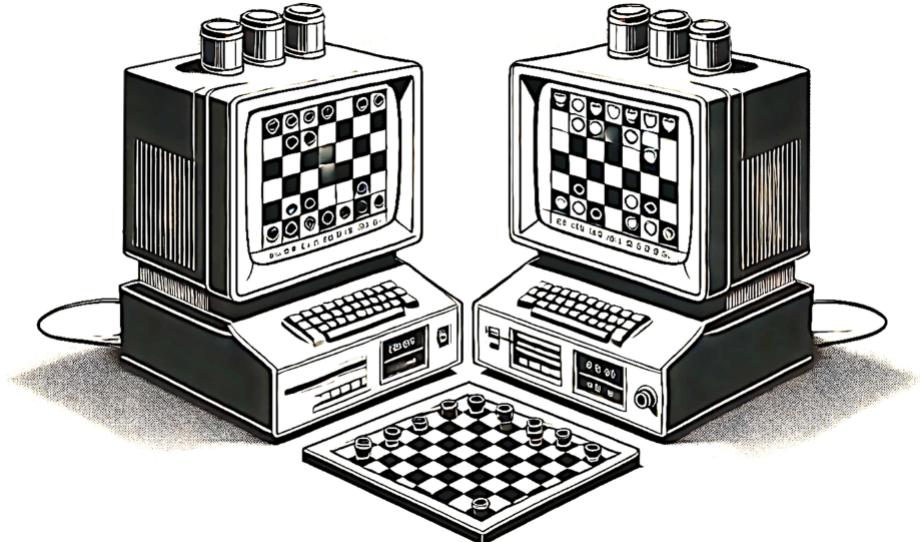
Les fondations du machine learning

Le machine learning est une branche de l'intelligence artificielle qui consiste à donner à une machine la capacité d'apprendre plutôt que de la programmer de façon explicite.

C'est ce qu'a fait Arthur Samuel, l'inventeur du machine learning, en 1959, lorsqu'il a réalisé le premier programme de jeu de Dame "intelligent". Ce programme était capable d'apprendre à jouer sans qu'aucune stratégie d'attaque ou de défense implémentée dans son système.

L'approche d'Arthur Samuel consistait à mettre face à face deux machines et à les faire jouer l'une contre l'autre avec des stratégies initialement aléatoires, puis à récompenser celle qui gagnait chaque partie avec un système de points. Au fil des parties, les programmes découvraient les situations qui leur faisaient gagner des points et celles leur en faisaient perdre. Ainsi, ils apprenaient d'eux-mêmes les meilleures stratégies pour gagner au jeu de dames.

Ce fut un succès!



Aujourd'hui, le machine learning est la technologie qui fait fonctionner ChatGPT, les voitures autonomes, et tous les systèmes dits "intelligents" depuis les trente dernières années. On peut dire que le machine learning est véritablement la discipline au cœur de l'IA.

Okay, mais comment ça fonctionne ?

L'idée centrale du machine learning est d'apprendre à partir de données. Pour cela, il existe trois grandes méthodes d'apprentissage :

Apprentissage Supervisé

Apprentissage Non-Supervisé

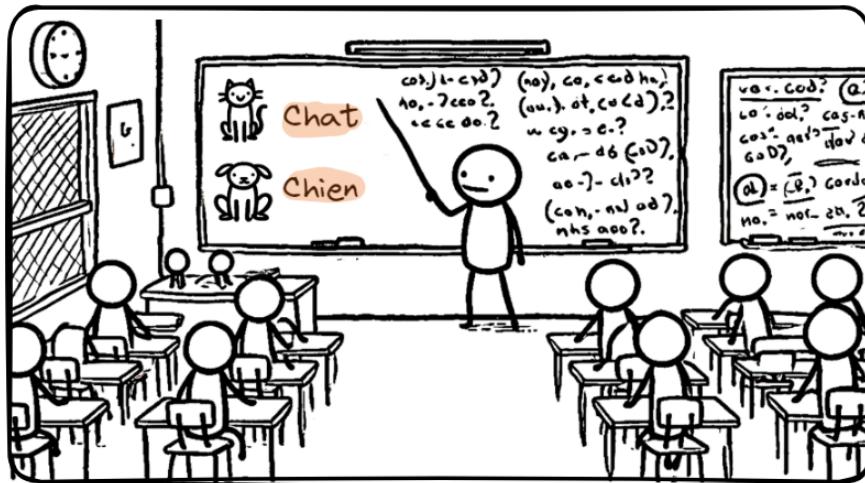
Apprentissage par Renforcement

Apprentissage supervisé

L'apprentissage supervisé est la branche la plus populaire du machine learning.

Comme son nom l'indique, il consiste à superviser l'apprentissage de la machine, de la même manière qu'un professeur supervise l'apprentissage de ses élèves en leur montrant des exemples de questions / réponses qu'ils doivent apprendre.

Que voyez-vous ici? réponse : un chat



La Machine étudie les exemples qu'on lui montre, et grâce à un système d'auto-évaluation et d'auto-amélioration que nous verrons dans ce livre, elle parvient à apprendre à réaliser la tâche qu'on lui demande d'effectuer.

Selon la nature de cette tâche, on distingue deux types de problèmes :

Les régressions	Les classifications
Lorsqu'on cherche à prédire une quantité :	Lorsqu'on cherche à prédire une classe :
	
Exemples : prix d'une maison, durée d'un trajet, température, poids, ...	Exemples : un animal, un état de santé, une couleur, un type de véhicule, ...

Voyons tout de suite un exemple de chacun.

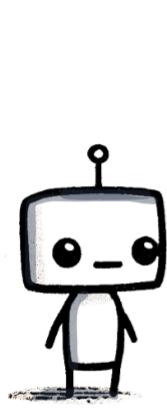
Régression

Imaginez que vous souhaitez prédire le prix d'une maison en fonction de sa surface habitable. Pour ce faire, on montre des exemples de maisons à notre machine.

Voici une maison, sa surface habitable est de 100 m² et son prix est de 252 000 euros.

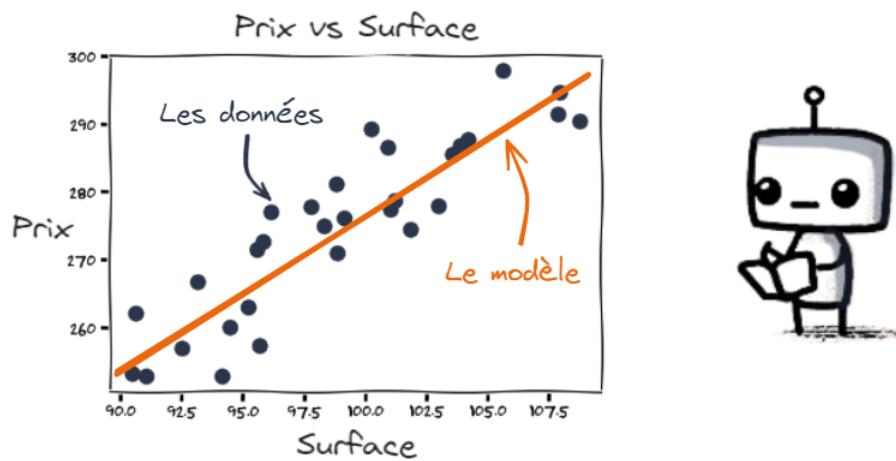


Dans l'apprentissage supervisé, ces exemples de questions/réponses sont présentés à la machine sous forme de jeu de données (X, y) , où X représente les variables d'entrée, et y la sortie attendue.



	X	y
	House icon	92
	House icon	95
	House icon	98
	House icon	93
	House icon	100
	House icon	110
		259K
		237K
		272K
		234K
		252K
		300K

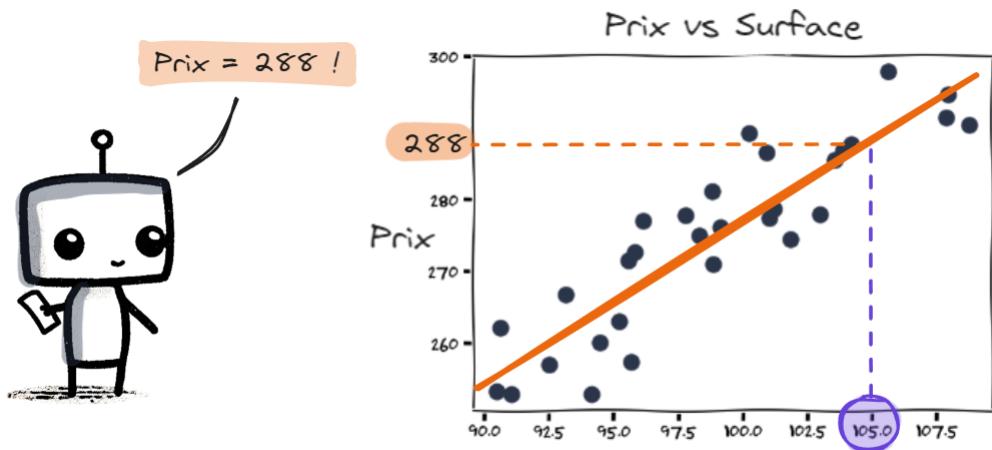
Grâce à ce jeu de données, la Machine est capable d'apprendre un modèle permettant de prédire la valeur de y en fonction de X . Pour ce faire, elle effectue une auto-évaluation en recherchant le modèle qui lui offre les meilleures performances par rapport au jeu de données fourni.



Une fois ce modèle développé, il est possible de s'en servir pour faire de futures prédictions. Par exemple, si nous avons une nouvelle maison, dont la surface habitable est de 105 m^2 :



Alors notre machine peut utiliser notre modèle pour prédire le prix de cette nouvelle maison.



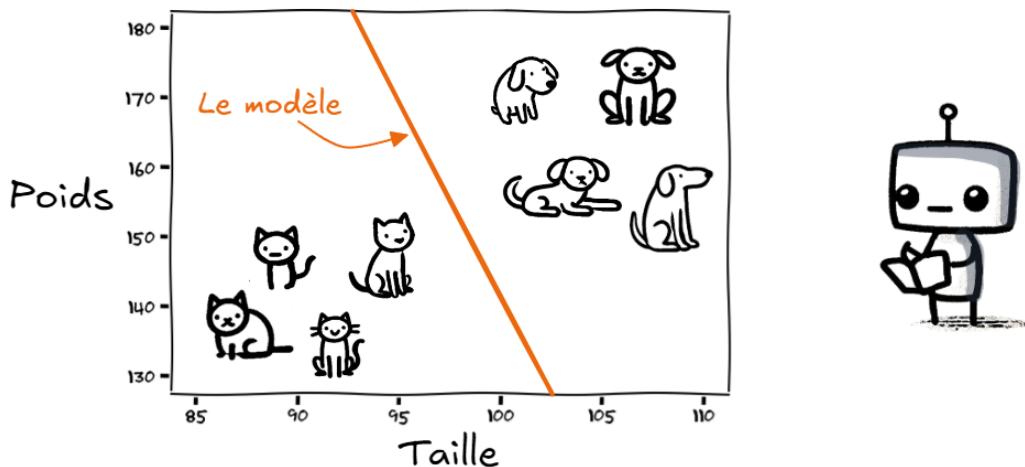
Classification

La logique est similaire pour prédire une classe. Imaginez que vous souhaitez apprendre à reconnaître un chien ou un chat en vous basant sur sa taille et son poids. Pour cela, il est nécessaire de fournir à la machine un jeu de données (X, y) afin qu'elle s'entraîne dessus.



	X		y
	Taille	Poids	
	92	133	Chat
	105	170	Chien
	87	135	Chat
	103	155	Chien
	107	150	Chien

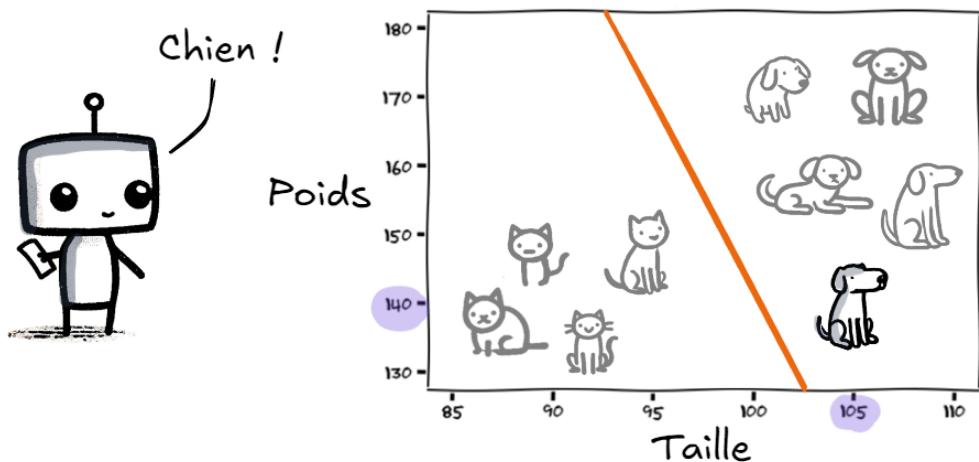
En étudiant ces données, la machine trouve le modèle qui sépare au mieux les deux classes de points, c'est-à-dire celui qui ne commet aucune erreur sur le jeu de données qui lui a été présenté.



Avec ce modèle, la machine peut désormais faire de nouvelles prédictions. Par exemple, voici ci-dessous, un animal qu'elle n'a jamais vu auparavant.



D'après le modèle développé à partir des données de référence, cet animal se situe du côté des chiens. La machine prédit donc qu'il s'agit d'un chien.



En résumé

Et voilà ! Vous savez désormais comment fonctionne l'apprentissage supervisé. En résumé, il s'agit de fournir des données (X, y) pour que la machine apprenne à prédire y en fonction de X , c'est-à-dire la relation $X \rightarrow y$.

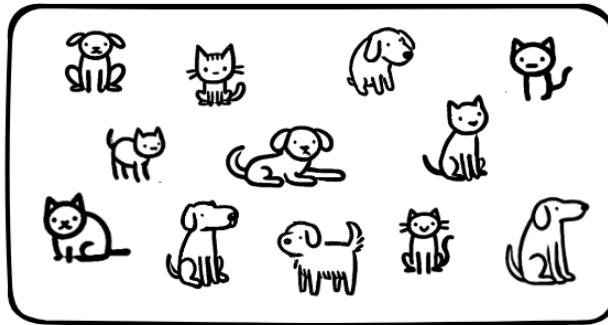
Nous aborderons plus en détail les modèles de régression et de classification dans les chapitres 2 et 3 de ce livre.

En attendant, il est temps de passer à la deuxième branche du machine learning, **l'apprentissage non supervisé**.

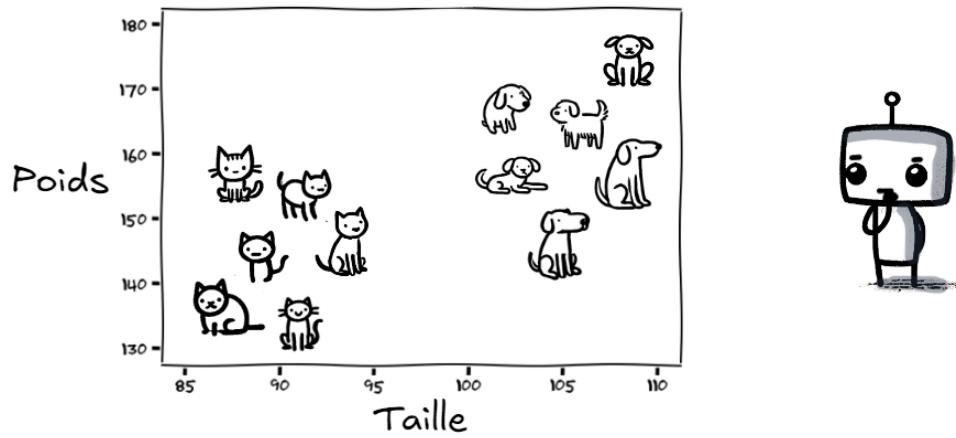
Apprentissage non supervisé

L'apprentissage non supervisé est la seconde branche très connue du machine learning. Contrairement à la méthode précédente, celle-ci consiste à laisser la machine apprendre par elle-même certaines structures présentes dans les données, sans la contraindre à apprendre une simple relation d'entrée / sortie $X \rightarrow y$.

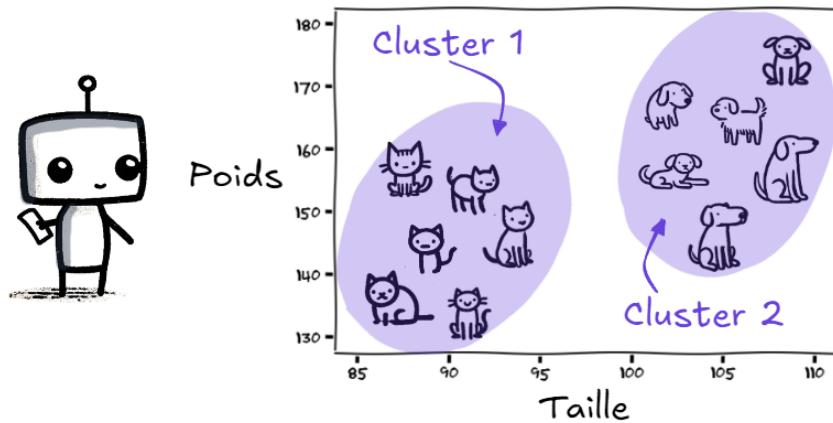
Par exemple, nous pouvons lui présenter les groupes d'animaux vus précédemment, sans préciser s'il s'agit de chats ou de chiens, et lui demander simplement de les regrouper selon leur ressemblance. Autrement dit, nous fournissons uniquement les attributs X , sans indiquer la sortie y attendue.



En examinant simplement les attributs X (tels que la taille, le poids, l'apparence, etc) la machine remarquera d'elle-même que certains animaux se ressemblent par leurs caractéristiques.

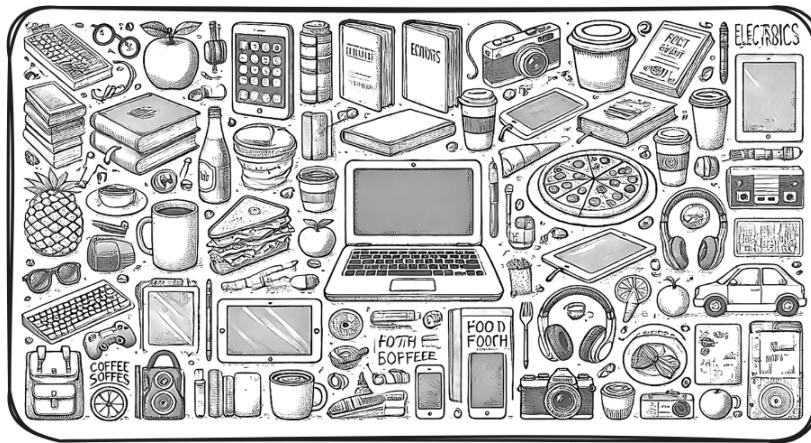


Ainsi, elle sera capable de les regrouper, sans pour autant savoir s'il s'agit de chats ou de chiens. En fait, elle n'en aura pas la moindre idée, car nous ne l'aurons pas contrainte à apprendre ce genre de choses.

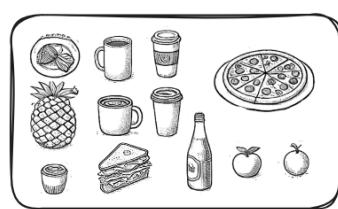
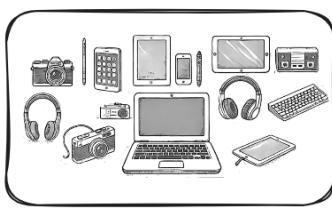
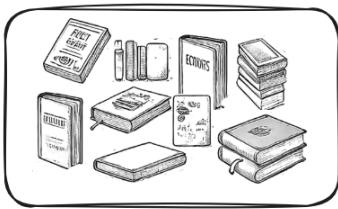


Ce type d'application porte le nom de **clustering**. Il s'agit de l'une des nombreuses applications de l'apprentissage non supervisé et démontre parfaitement son utilité. En effet, dans la pratique, il peut être très intéressant de laisser la machine proposer sa propre manière de faire les choses, plutôt que de la contraindre, de manière supervisée, à reproduire notre façon de faire.

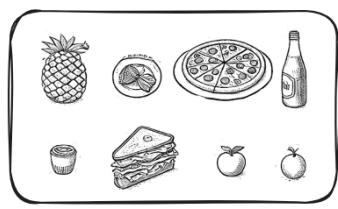
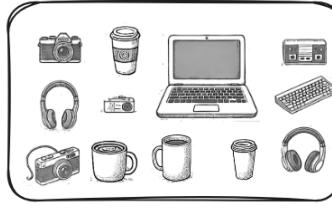
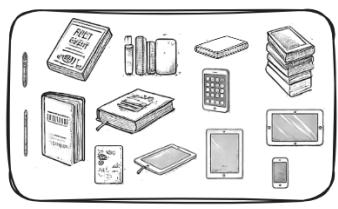
Prenez par exemple un panier d'objets achetés par différents consommateurs sur un site de vente en ligne.



En tant qu'êtres humains, nous aurions tendance à vouloir regrouper ces objets en trois catégories : livres, appareils électroniques et nourriture.



Cependant, en examinant tous les objets de manière non supervisée, la machine pourrait découvrir une autre façon de les classer, en identifiant des clusters liés aux préférences des clients, aux prix, aux marques, etc. Par exemple, elle pourrait regrouper les tablettes graphiques avec les livres, car on peut réaliser des activités similaires avec ces deux objets.

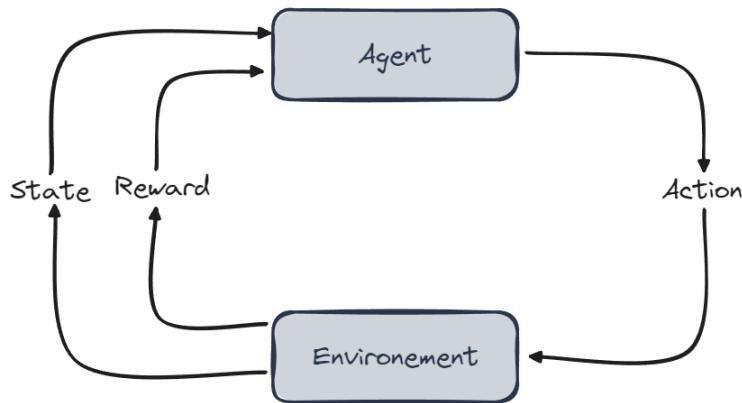


Nous aborderons plus en détail le clustering dans le chapitre 4 de ce livre, ainsi qu'une autre application de l'apprentissage non supervisé, à savoir la détection d'anomalies, dans le chapitre 5.

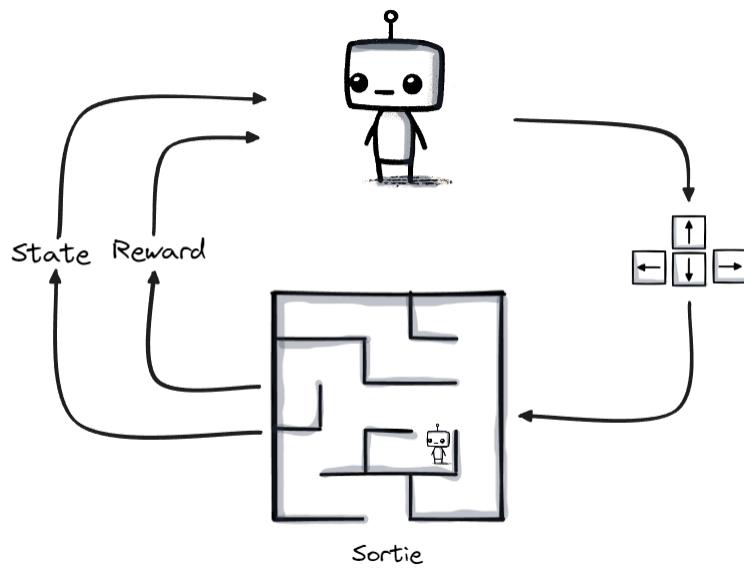
Apprentissage par renforcement

L'apprentissage par renforcement est la troisième branche du machine learning. Cette méthode, très différente des deux autres, consiste à laisser la machine générer ses propres données issues de son expérience et à apprendre à partir de celles-ci.

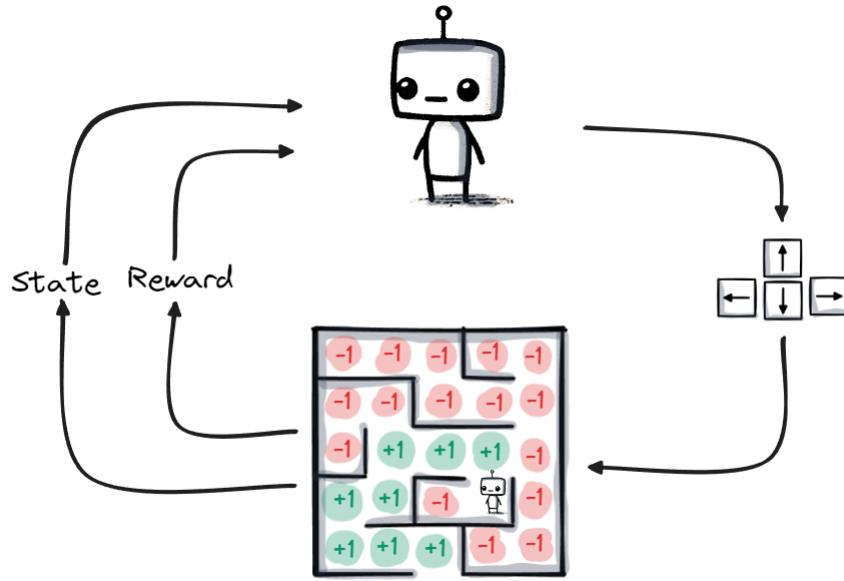
L'idée est de créer un agent, libre d'entreprendre des actions au sein d'un environnement, et de récompenser cet agent lorsque les actions qu'il choisit de prendre le mènent au résultat souhaité.



Par exemple, l'agent pourrait être un petit personnage et l'environnement, un labyrinthe dont il doit trouver la sortie. Pour cela, il est libre de se déplacer dans toutes les directions qu'il souhaite.



À chaque fois qu'il se déplace, l'agent reçoit une récompense liée à la pertinence de son choix. Par exemple, se rapprocher de la sortie lui accorde un point, tandis que s'en éloigner lui en fait perdre un.



À force d'essais et d'erreurs, l'agent génère ses propres expériences et apprend de celles-ci, un peu comme un enfant qui apprend à faire du vélo. Pour ce type d'application, les apprentissages supervisé et non supervisé ne sont daucun secours. En effet, personne n'a jamais appris à faire du vélo en regardant les autres en faire. Il faut pratiquer, tomber, puis recommencer

L'apprentissage par renforcement trouve ainsi des applications dans de nombreux domaines avancés, tels que la robotique, les voitures autonomes, la prise de décision et bien d'autres encore. Nous aborderons ces sujets plus en détail dans le chapitre 6.

À votre avis ?

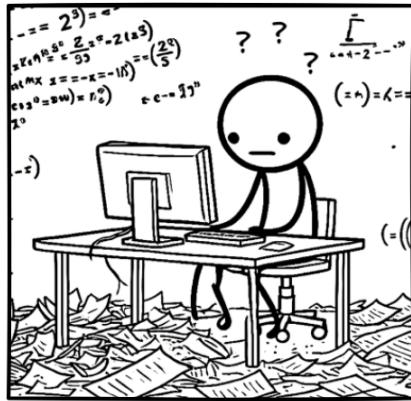
Avant d'aller plus loin, voici une petite question pour vous : à votre avis, quelle méthode d'apprentissage Arthur Samuel a-t-il employée pour développer son jeu de dames?

Supervisée? Non supervisée? Ou par renforcement?

Pour résumer

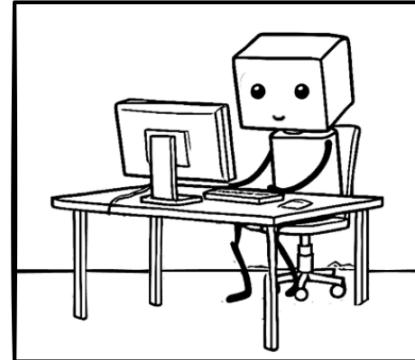
L'approche du machine learning diffère de celle de la programmation classique, qui consiste généralement à définir nous-mêmes le fonctionnement des modèles.

Cette capacité à apprendre à partir de données explique pourquoi le machine learning surpassé aujourd'hui les performances des modèles traditionnels dans certaines applications, comme la vision par ordinateur ou les chatbots.



En effet, il est très difficile pour un développeur de créer manuellement un modèle mathématique capable de reconnaître un objet sur une photo. Il faudrait programmer une infinité de modèles géométriques et trigonométriques pour détecter des formes élémentaires telles que des droites, des cercles, des triangles, puis assembler ces formes afin de tenter de comprendre à quoi elles correspondent. Sans oublier de prendre en compte la luminosité, les ombres, la perspective... Bref, un vrai cauchemar.

Le machine learning, quant à lui, permet d'automatiser cette tâche de recherche en trouvant lui-même le meilleur modèle sur la base des données de références qu'on lui fournit. Son seul objectif est de "bien faire" sur les données qui lui sont présentées. C'est pourquoi il est si performant.



Maintenant que vous comprenez le fonctionnement de base du machine learning, il est temps de se plonger dans chacune des grandes techniques que nous avons mentionnées :

- Chapitre 2 : Régression
- Chapitre 3 : Classification
- Chapitre 4 : Clustering
- Chapitre 5 : Détection d'anomalies
- Chapitre 6 : Apprentissage par renforcement
- Chapitre 7 : Deep learning et generative AI

Régression



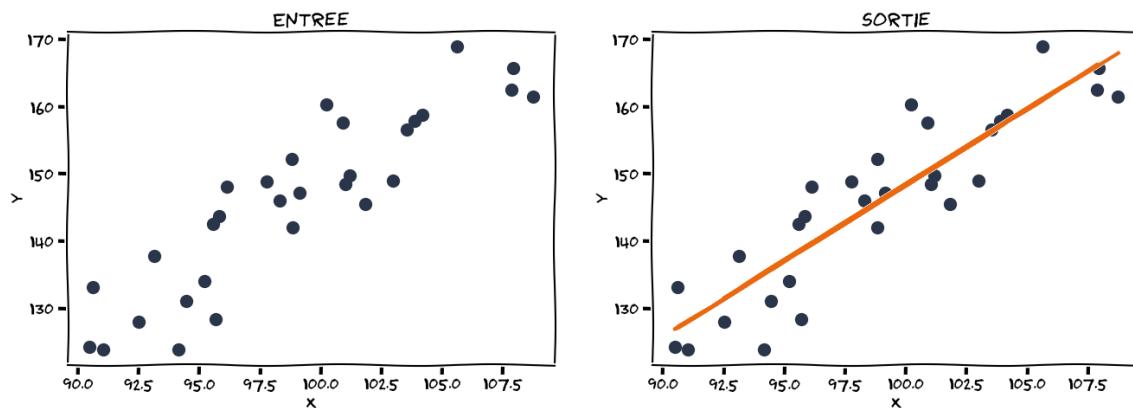
Régression

Dans ce chapitre, vous allez apprendre à développer des modèles de **régression**. Comme vu dans le chapitre 1, ces modèles sont utiles chaque fois que vous souhaitez prédire une variable quantitative y à partir de données d'entrée X .

Par exemple :

- Le prix d'une maison
- Le temps de trajet d'un taxi
- Le nombre de visiteurs quotidiens sur un site Internet
- Le chiffre d'affaires d'une entreprise pour l'année prochaine
- etc.

Pour rappel, ces modèles sont créés de manière **supervisée**, c'est-à-dire que l'on montre à la machine des exemples de données (X, y) afin qu'elle développe un modèle permettant de relier $X \rightarrow y$.

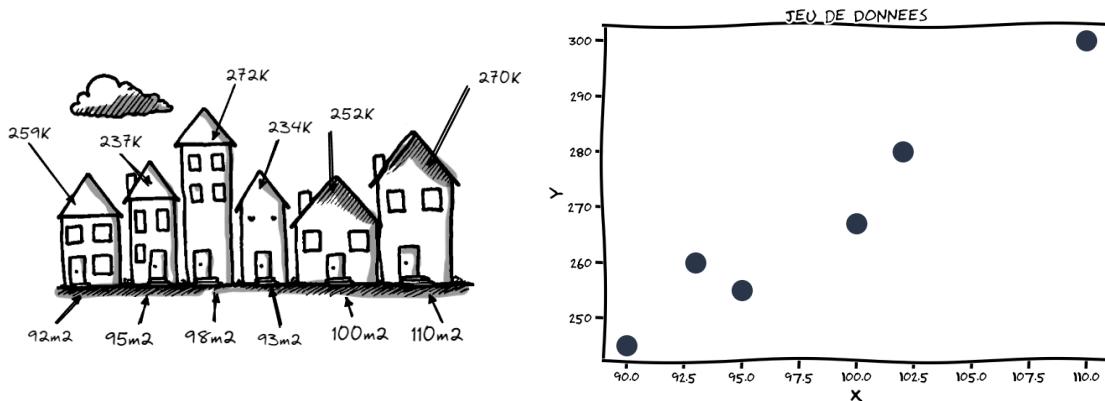


Mise en situation

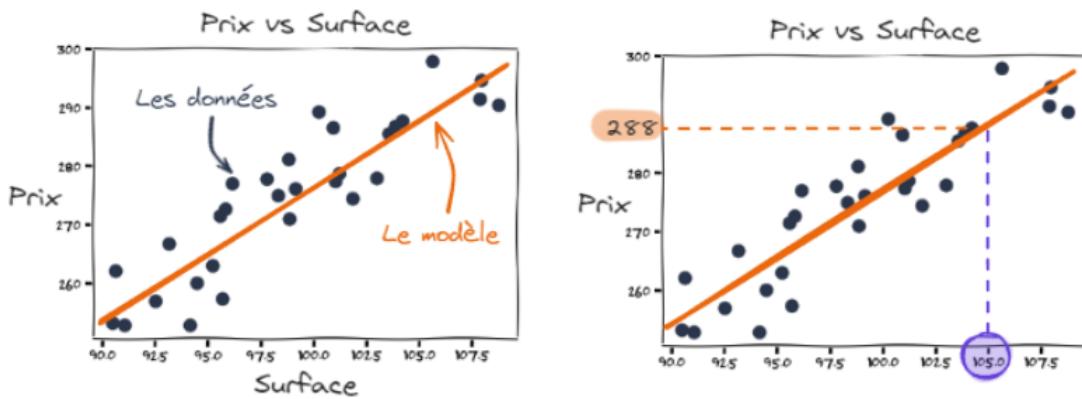
Imaginez que vous souhaitez développer un modèle capable de prédire le prix d'une maison en fonction de sa surface habitable.



Pour cela, vous disposez d'une base de données de référence qui contient plusieurs exemples de maisons, avec pour chacune leur prix y ainsi que leur surface habitable X . En affichant ces points sur un graphique, on obtient le résultat suivant.



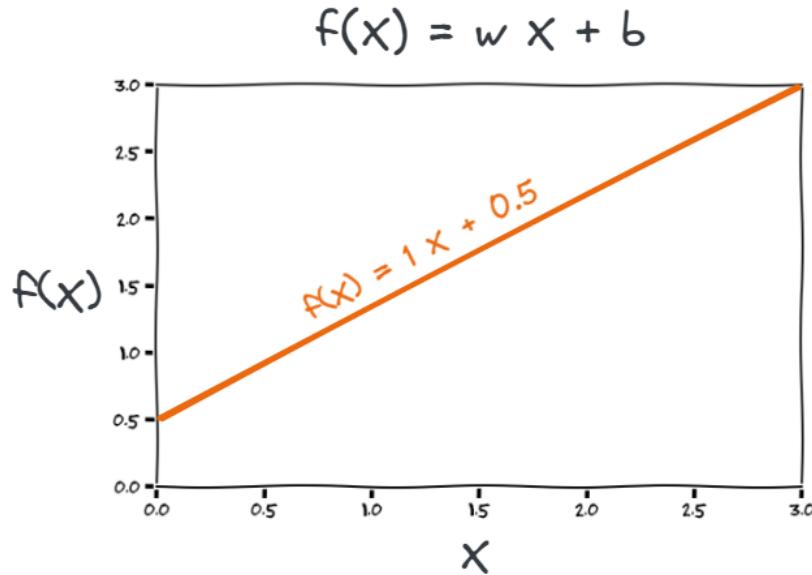
Comme nous l'avons vu dans le chapitre 1, l'essence même de l'apprentissage supervisé est de développer un modèle qui représente au mieux ces données, pour ensuite l'utiliser afin de faire de nouvelles prédictions.



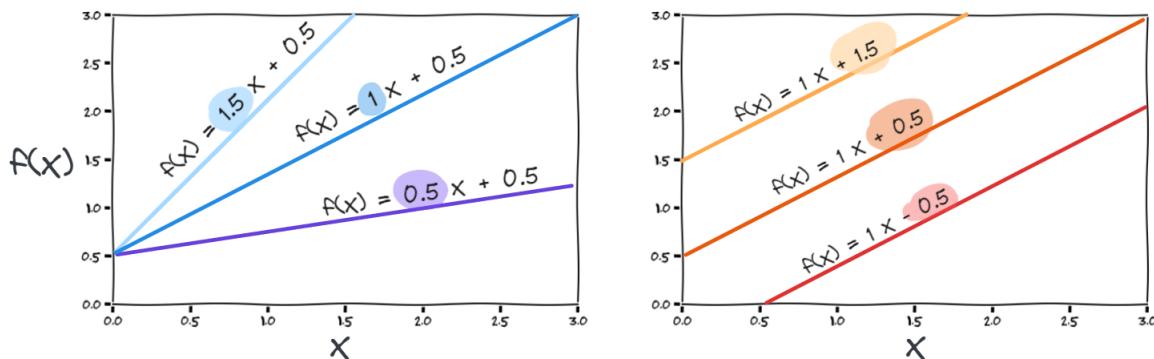
En machine learning, il existe de nombreux modèles permettant de réaliser ce genre de tâche. Dans ce chapitre, nous allons étudier le plus simple d'entre eux : le modèle de **régression linéaire**.

Le modèle de régression linéaire

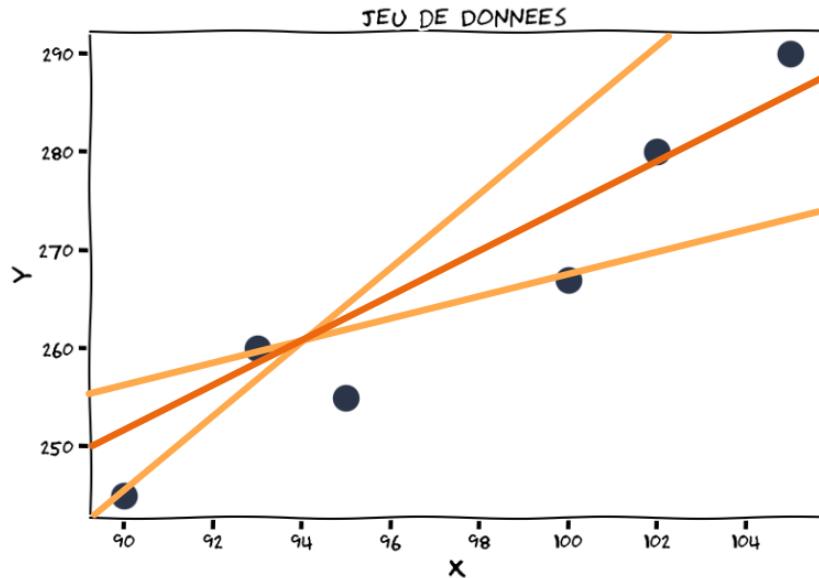
Le modèle de régression linéaire peut être décrit avec la formule bien connue $f(x) = wx + b$. Lorsque l'on trace cette fonction, on obtient une droite :



Dans ce modèle, les paramètres w et b contrôlent l'allure de la droite : w détermine son inclinaison, et b son ordonnée à l'origine.

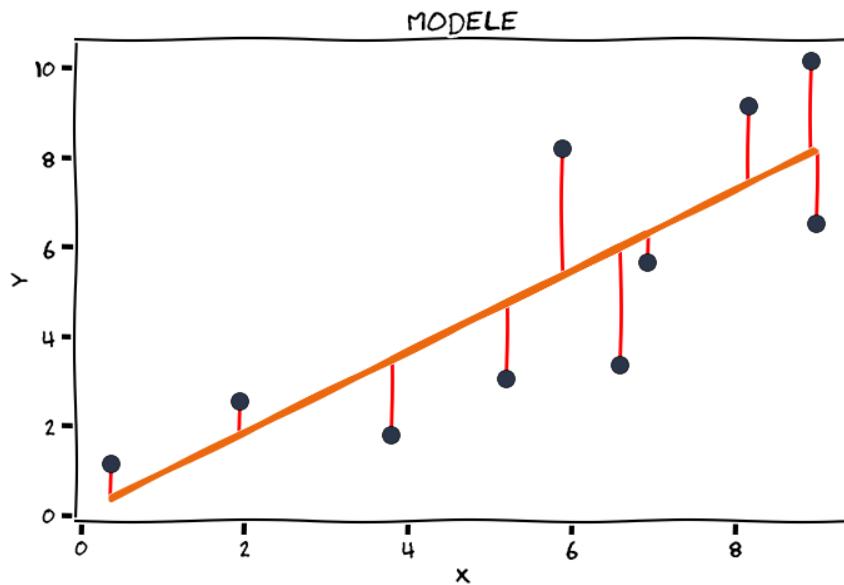


Notre objectif est de trouver les paramètres w et b qui permettent au modèle de s'ajuster au mieux au jeu de données (X, y)



Normalement, ce serait le travail d'un développeur de déterminer ces paramètres, mais en machine learning, c'est la machine elle-même qui va s'en charger, grâce à un algorithme d'optimisation qui va chercher à **minimiser** les erreurs entre le modèle et les points de données.

Tout commence donc par la mesure de ces erreurs, représentées ici en rouge. C'est ce qu'on appelle la **fonction coût**.



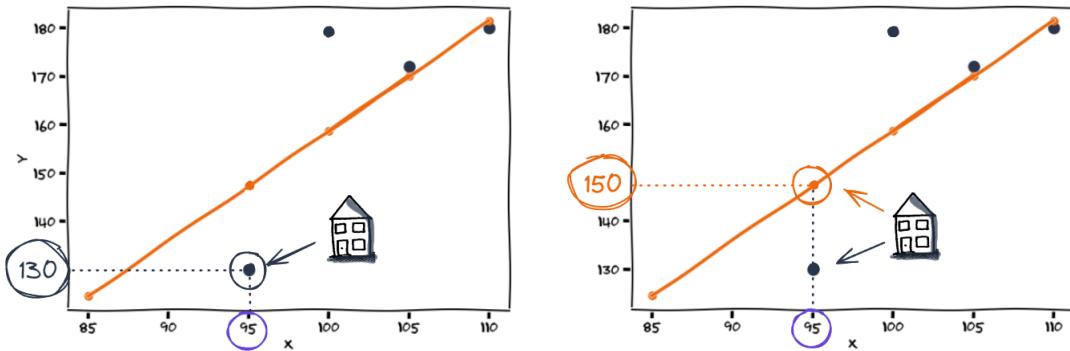
La fonction coût

Calculer la fonction coût est très simple : il s'agit de mesurer l'erreur entre chaque point de données et notre modèle, puis de faire la moyenne de toutes ces erreurs.

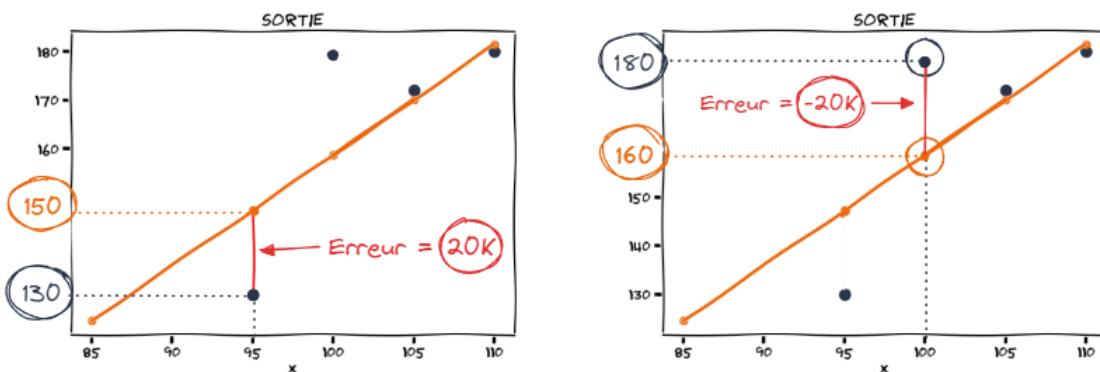
Pour ce faire, on commence par calculer la différence entre la prédiction du modèle pour chaque point (i) et la valeur attendue $y^{(i)}$. On appelle cette différence le résidu $R^{(i)}$

$$R^{(i)} = f(x^{(i)}) - y^{(i)}$$

Par exemple, pour la première maison du jeu de données ($i = 1$) la surface habitable $x^{(1)} = 95$ et le prix $y^{(1)} = 130$. Si le modèle prédit $f(x^{(1)}) = 150$ alors le résidu $R^{(1)} = f(x^{(1)}) - y^{(1)} = 150 - 130 = 20$. Autrement dit, le modèle a fait une erreur de prix de 20 000 euros.



Avant de faire la somme de tous ces résidus, il est important de s'assurer qu'ils ne puissent pas s'annuler les uns avec les autres. En effet, que se passe-t-il si l'on additionne les deux résidus de signes opposés ci-dessous? Ils s'annulent...



Pour éviter cela, on élève chaque résidu au carré, ce qui permet de n'obtenir que des résidus positifs. Cela nous donne ce qu'on appelle la **distance euclidienne**, notée $D^{(i)}$

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

Pour finir, on calcule la somme, puis la moyenne de toutes ces erreurs. Cela nous donne la **perte globale** du modèle (*Loss* en anglais).

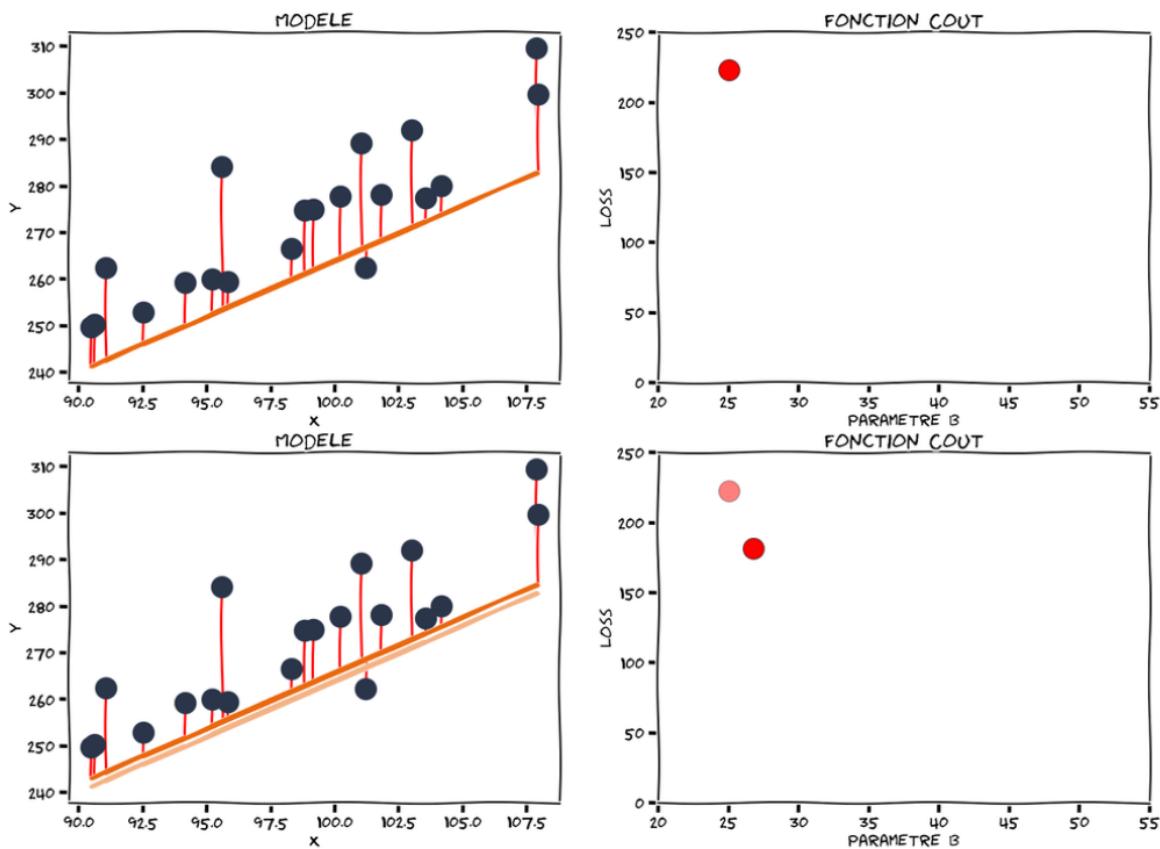
$$Loss = \frac{1}{2m} \sum_{i=1}^m \left(y^{(i)} - f(x^{(i)}) \right)^2$$

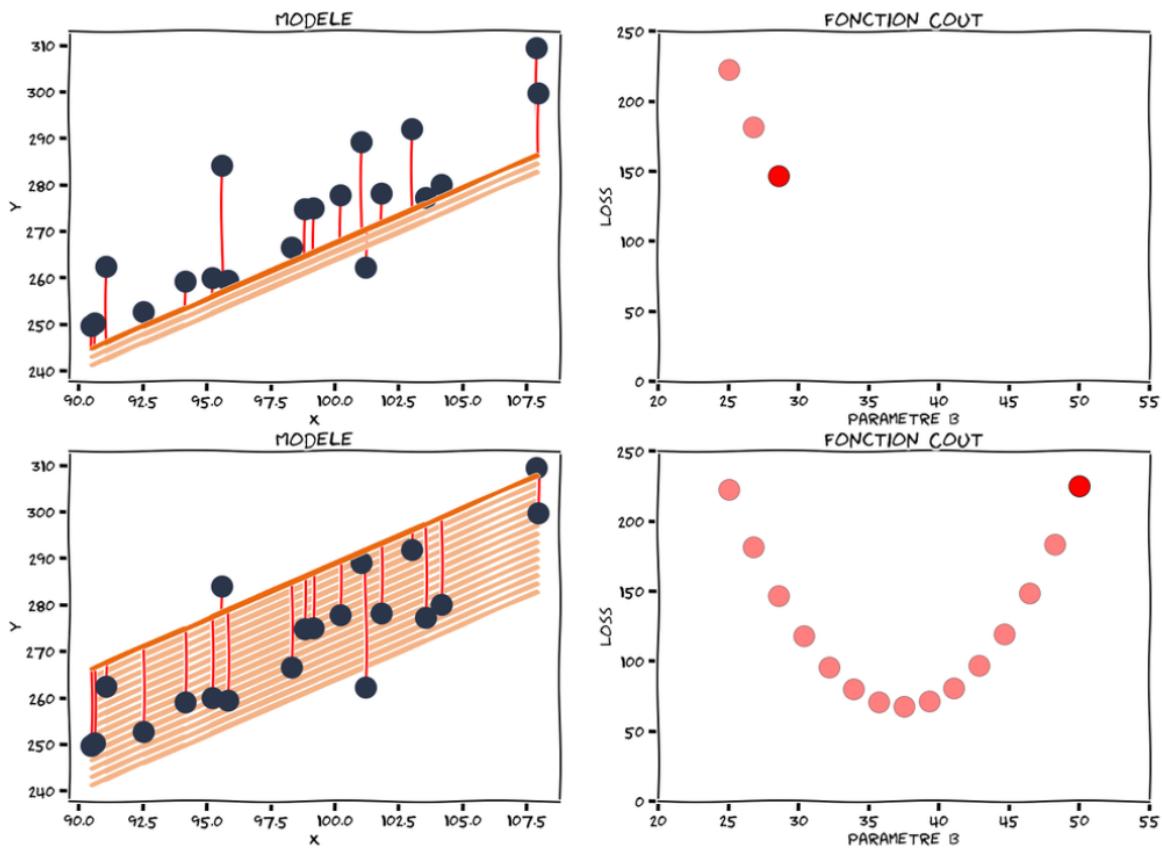
(*m* désigne le nombre total de points dans notre jeu de données)

Et voilà ! C'est ainsi que l'on obtient la fonction coût.

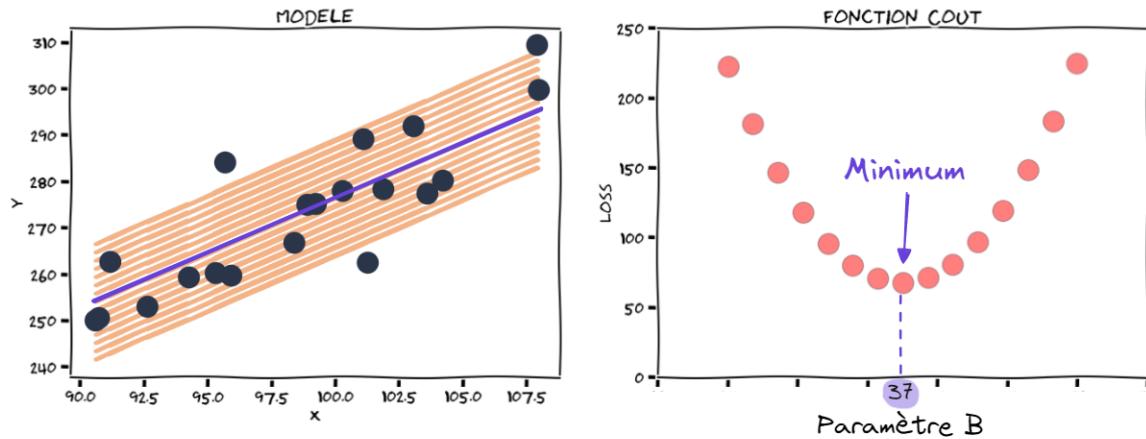
Allure de la fonction coût

Lorsqu'on fait varier les paramètres w et b de notre modèle, on constate que la fonction coût prend la forme d'une parabole. Le graphique ci-dessous illustre ce qui se produit lorsqu'on calcule la fonction coût pour différents modèles en faisant varier la valeur de b

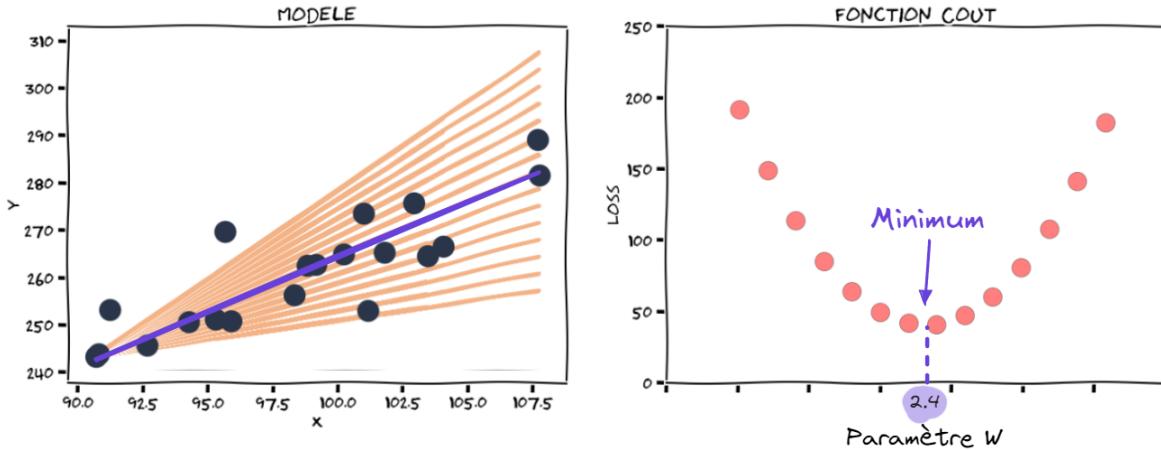




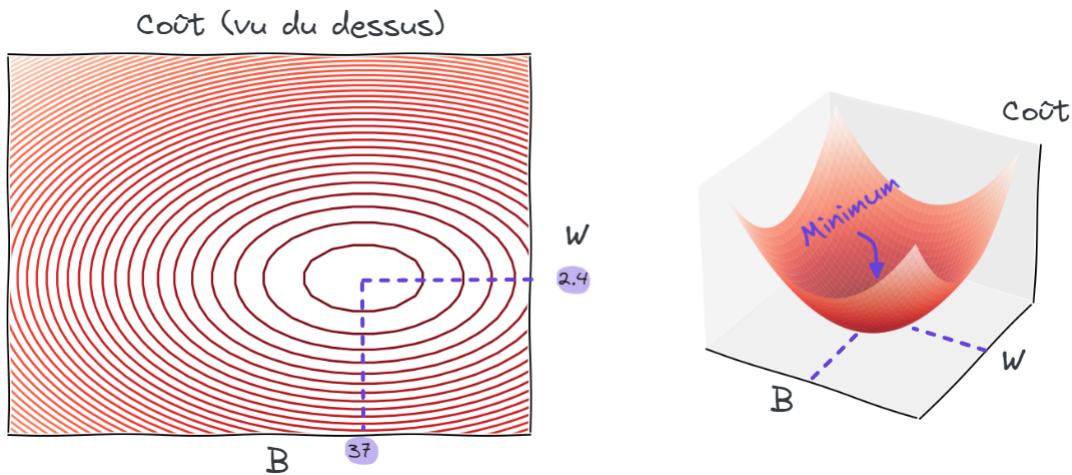
Dans cet exemple, on voit clairement que le minimum de la fonction coût est atteint lorsque b avoisine la valeur 37.



La même chose se produit lorsqu'on fait varier l'inclinaison de notre modèle, c'est-à-dire le paramètre w . Ici, on observe que le modèle fait les plus petites erreurs lorsque $w = 2.4$



Il est possible de visualiser ces deux graphiques en les combinant dans un graphique 3D. On obtient alors une surface en forme de bol, au fond de laquelle se trouve le minimum de la fonction coût, ainsi que les coordonnées (w, b) permettant d'y accéder.

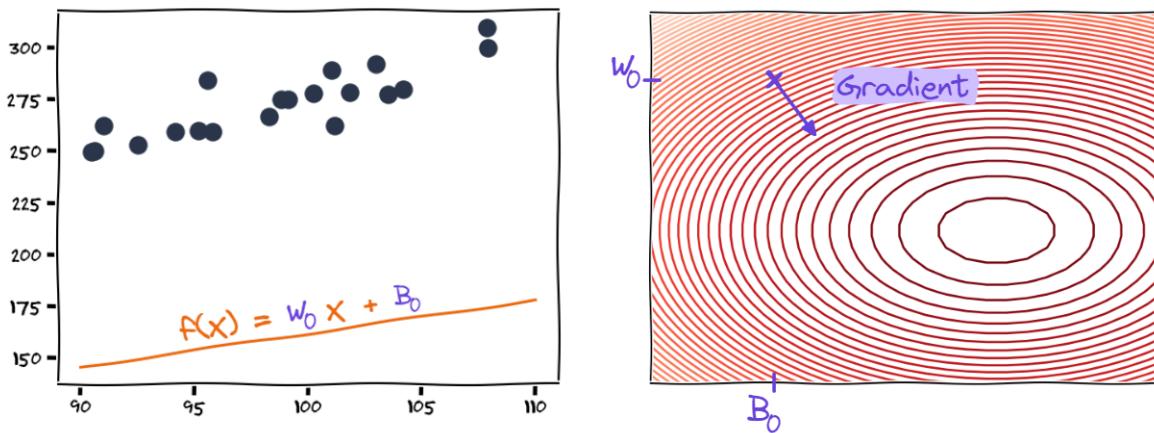


À première vue, il semble donc simple de trouver ce minimum. Il suffit de regarder où il se situe. Mais cela n'est possible que parce que nous avons créé cette carte en testant de nombreuses valeurs de (w, b) . Notre machine quant à elle, ignore complètement où se situe ce minimum, car elle ne dispose pas de cette carte. C'est pourquoi nous allons développer un algorithme qui lui permettra de trouver ce minimum automatiquement. Cet algorithme porte le nom de **descente de gradient**

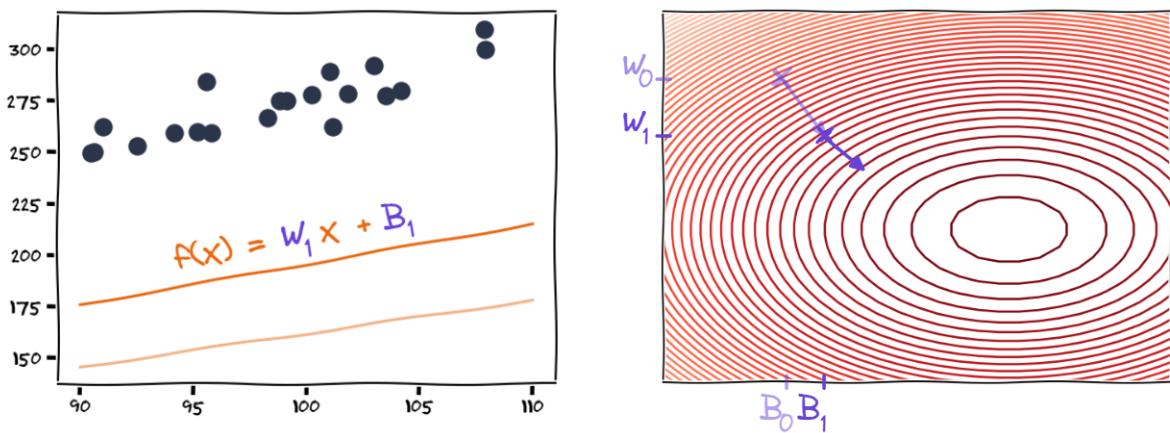
L'algorithme de la descente de gradient

La **descente de gradient** est l'un des algorithmes d'apprentissage les plus utilisés en machine learning. Il consiste à calculer le gradient de la fonction coût, c'est-à-dire comment celle-ci évolue lorsque w et b varient légèrement, pour ensuite faire un pas dans la direction où la fonction coût diminue. D'où le nom de descente de gradient.

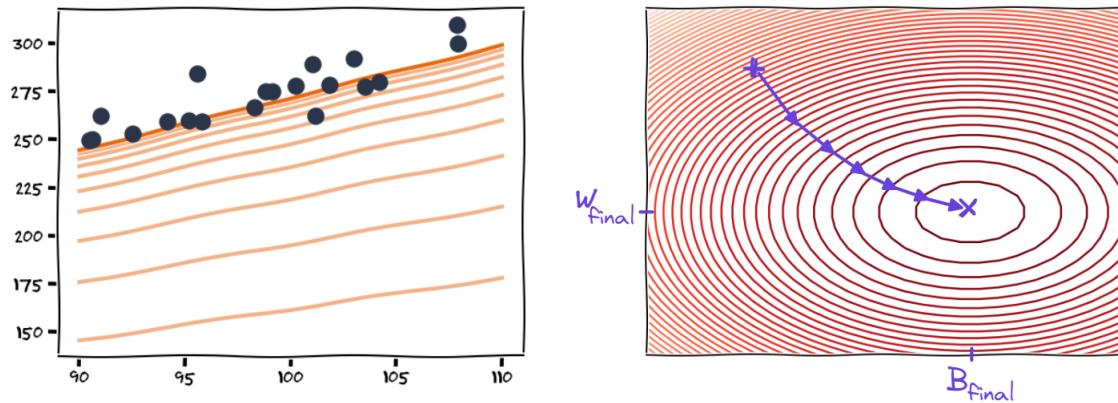
L'algorithme démarre avec des paramètres (w_0, b_0) choisis au hasard. Il en résulte un modèle initial, comme illustré ci-dessous. À partir du point (w_0, b_0) on calcule le gradient, c'est-à-dire la direction de la pente.



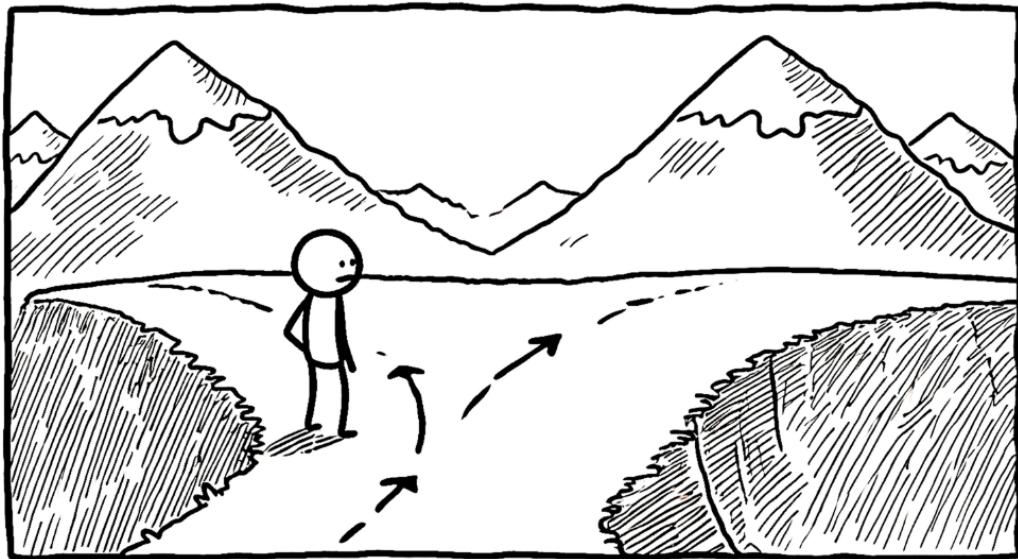
On effectue alors un pas dans la direction de la descente. Cela nous donne de nouvelles valeurs pour (w, b) , notées (w_1, b_1) . Avec ces nouvelles valeurs, nous obtenons également un nouveau modèle, qui se rapproche davantage des données (X, y) . Ensuite, on calcule à nouveau le gradient au nouveau point (w_1, b_1) , ce qui nous permet de poursuivre notre descente.



En continuant ainsi, on converge vers le minimum de la fonction coût, ce qui nous donne le modèle final.



Et voilà! C'est ainsi que fonctionne l'algorithme de la descente de gradient. On peut faire l'analogie avec un randonneur perdu en pleine montagne, dont le but est de rejoindre sa voiture, garée au point le plus bas de la vallée. Il ne dispose pas de carte de la région, alors pour s'assurer de trouver son chemin, il va constamment suivre la direction qui descend. Cette stratégie fonctionne tant qu'il n'y a qu'un seul creux dans la vallée, comme c'est le cas pour la fonction coût.



D'un point de vue mathématique, l'algorithme de la descente de gradient est décrit par les formules suivantes :

$$w_{t+1} = w_t - \eta \times \frac{\partial L}{\partial w_t}$$

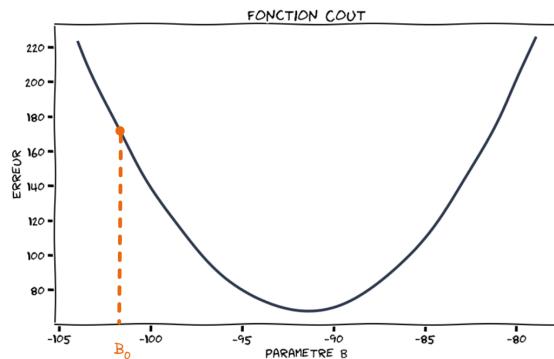
$$b_{t+1} = b_t - \eta \times \frac{\partial L}{\partial b_t}$$

Explication :

Dans ces formules, on met à jour la valeur de w et b en prenant leur valeur actuelle, et en soustrayant à chacun la valeur du gradient, notée respectivement $\frac{\partial L}{\partial w_t}$ et $\frac{\partial L}{\partial b_t}$

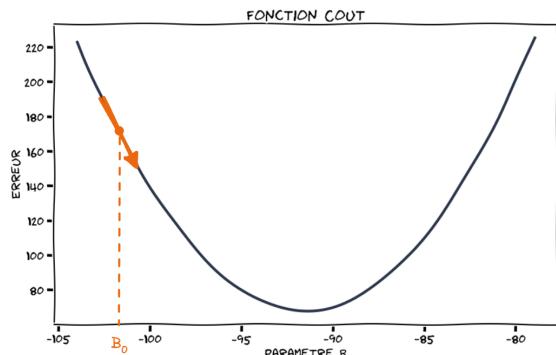
(η représente une vitesse d'apprentissage. Dans notre cas, nous allons poser $\eta = 1$

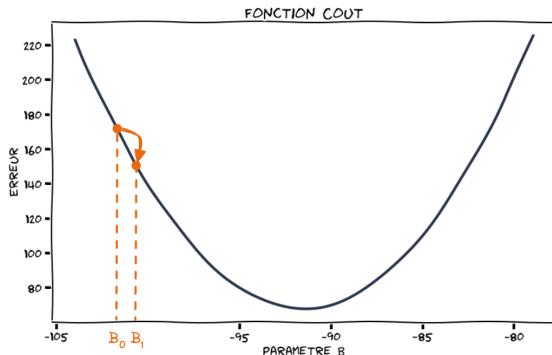
Prenons pour exemple le paramètre b :



Nous débutons l'algorithme d'apprentissage avec une valeur initiale de b_0 choisie au hasard. Ici, $b_0 = -102$.

En calculant le gradient $\frac{\partial L}{\partial b}$, on obtient une valeur négative (ce qui est logique, car la fonction diminue lorsque l'on augmente la valeur de b). Pour ne pas rentrer dans des calculs inutiles, supposons que $\frac{\partial L}{\partial b} = -2$





Ainsi, la formule de la descente de gradient donne :

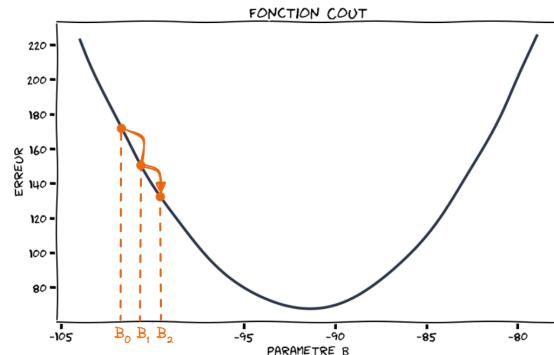
$$b_1 = b_0 - \eta \times \frac{\partial L}{\partial b_0}$$

$$b_1 = b_0 - 1 \times (-2)$$

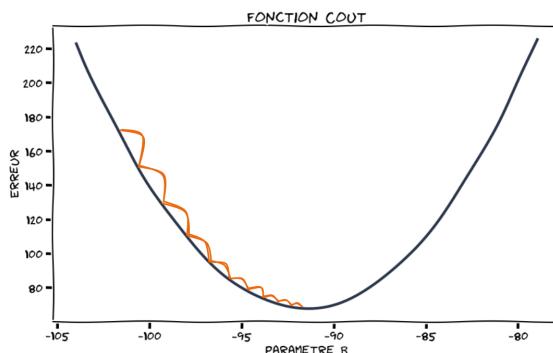
$$b_1 = b_0 + 2$$

b_1 est donc supérieur à b_0

On répète ainsi l'algorithme...



...jusqu'à convergence.



Et voilà ! C'est ainsi que fonctionne l'algorithme de la descente de gradient, dont l'utilisation va bien au-delà de la régression linéaire. En effet, la descente de gradient est un algorithme d'optimisation largement exploité en deep learning, notamment pour entraîner des réseaux de neurones (dont nous parlerons dans le dernier chapitre de ce livre).

Vous savez désormais comment implémenter un modèle de régression linéaire. Voyons à présent comment le faire avec Python.

Implémentation Python

Avant toute chose, si vous n'avez jamais développé un programme en Python, je vous invite à consulter ma série de vidéos à ce sujet ([disponible ici](#)). Vous y apprendrez comment installer Python sur votre ordinateur et bien démarrer avec ce langage.

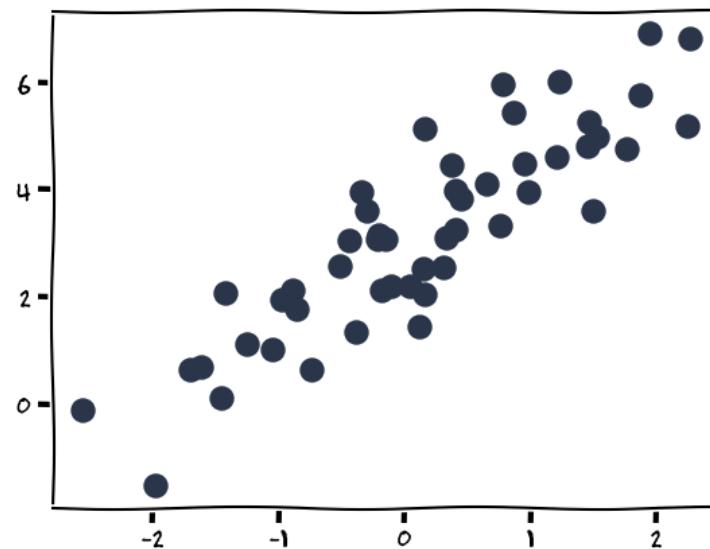
Sinon, c'est parti! 😊

Pour développer un modèle de régression linéaire avec Python, nous allons utiliser le package **scikit-learn**, très populaire pour faire du machine learning.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.linear_model import LinearRegression
```

Pour commencer, simulons des données avec la librairie Numpy. Nous avons ici 50 points, distribués en suivant une relation $y = 1.5x + 3$

```
1 np.random.seed(0)
2
3 m = 50
4 X = np.random.randn(m).reshape(-1, 1)
5 y = 1.5 * X + 3 + np.random.randn(m).reshape(-1, 1)
6
7 plt.scatter(X, y)
8 plt.show()
```

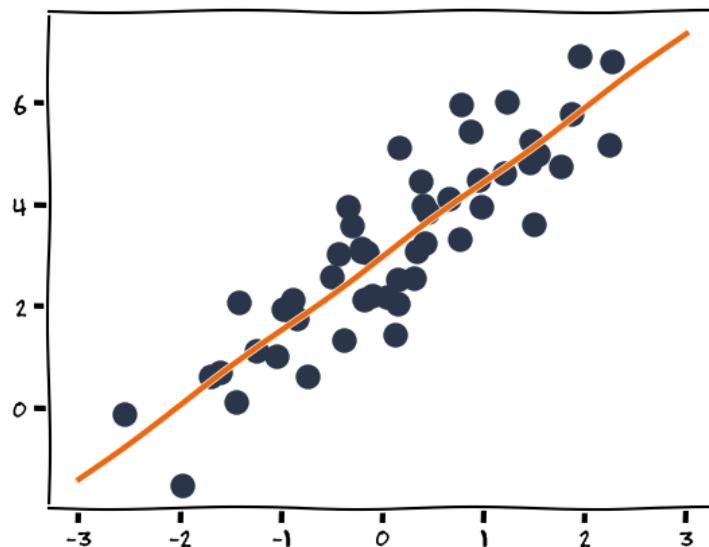


Pour créer notre modèle de machine learning, nous allons utiliser le modèle **LinearRegression** de scikit-learn. Ce modèle n'utilise pas la descente de gradient vue dans ce chapitre, mais une autre méthode d'optimisation, plus rapide, que je détaille sur ma chaîne YouTube *Machine Learnia*.

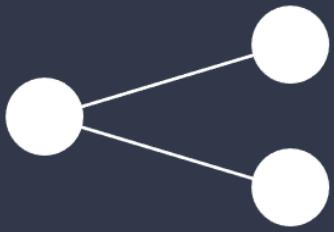
Une fois le modèle créé, nous utilisons la méthode **fit()** pour entraîner le modèle sur les données (X, y)

Une fois le modèle entraîné, il suffit de l'utiliser pour faire des prédictions à l'aide de la méthode **predict()**

```
1 model = LinearRegression()
2 model.fit(X, y)
3
4 new_data = np.linspace(-3, 3).reshape(-1, 1)
5 predictions = model.predict(new_data)
6
7 plt.scatter(X, y)
8 plt.plot(new_data, predictions)
9 plt.show()
```



Classification

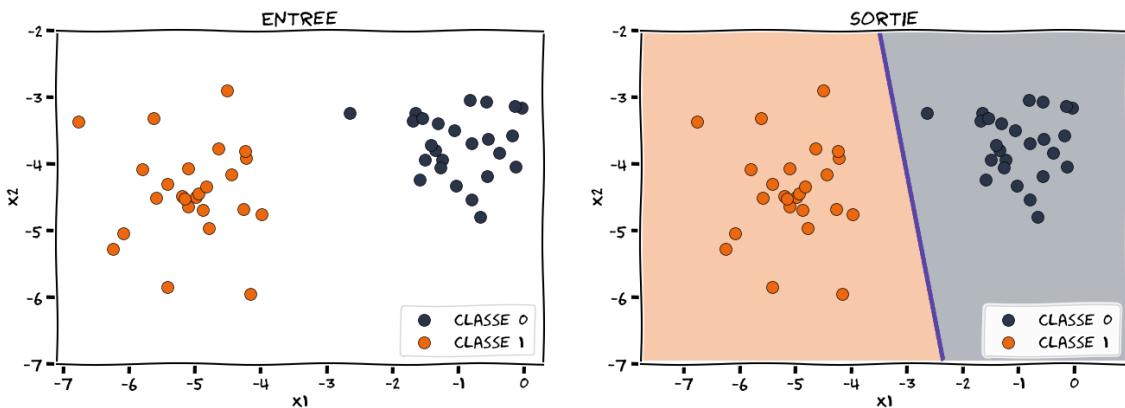


Classification

Dans ce chapitre, vous allez apprendre à développer des modèles de **classification**. Comme leur nom l'indique, ces modèles sont utilisés pour prédire la classe ou la catégorie à laquelle appartient un objet. Par exemple :

- Prédire si un email est un spam ou non (2 classes possibles)
- Déetecter si une cellule est cancéreuse ou non (2 classes possibles)
- Prédire le style de musique préféré d'une personne (environ 10 classes possibles)
- Reconnaître une plante et donner son nom (des milliers de classes possibles)
- etc.

Pour rappel, ces modèles sont créés de manière **supervisée**, c'est-à-dire que l'on montre à la machine des exemples de données (X, y) afin qu'elle développe un modèle permettant de relier $X \rightarrow y$, cela permet d'apprendre à séparer les différentes classes du jeu de données.

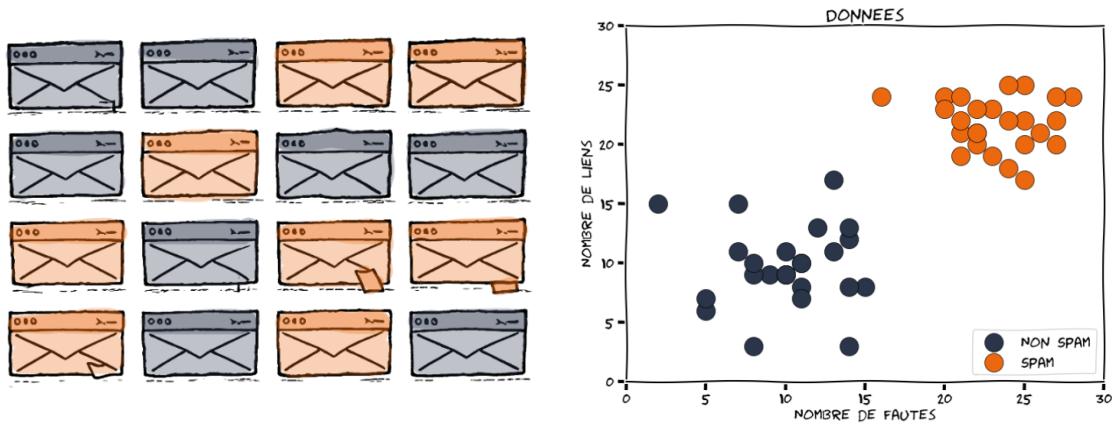


Mise en situation

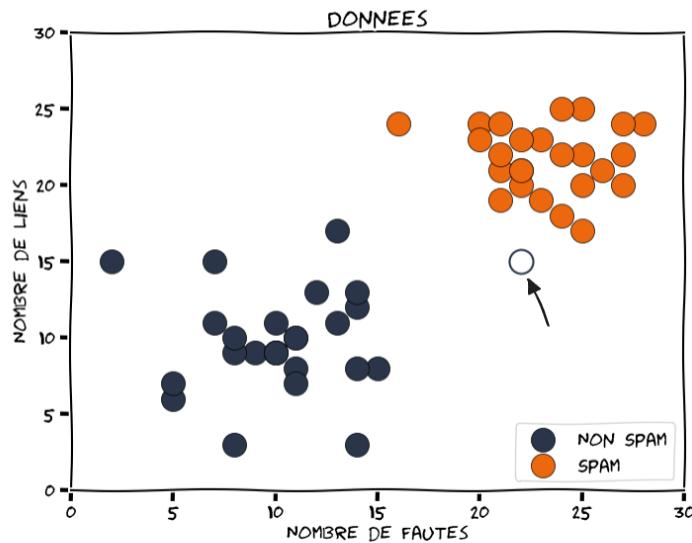
Imaginez que vous souhaitez prédire si un email est un spam ou non, en vous basant sur le nombre de liens et le nombre de fautes d'orthographe qu'il contient.



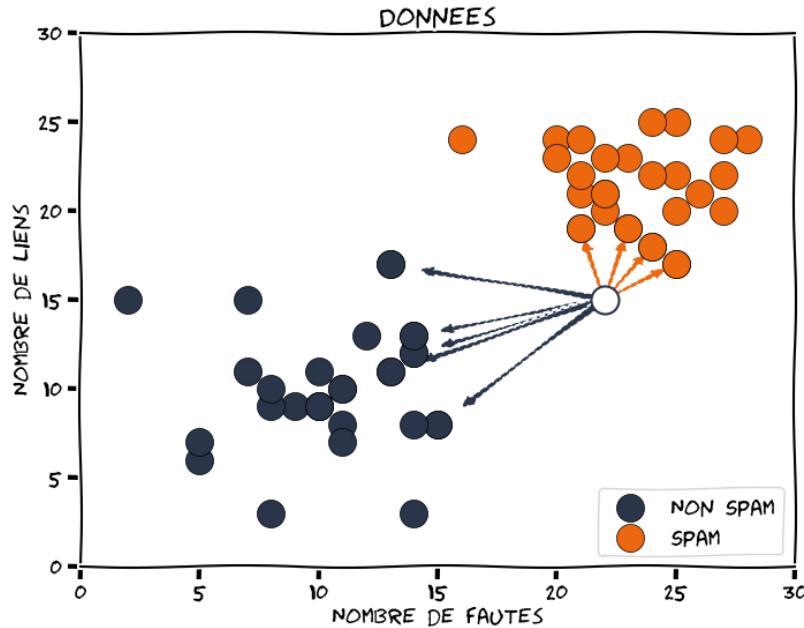
Pour cela, vous disposez d'un jeu de données d'emails correspondant à la tâche que vous cherchez à accomplir.



Notre email, quant à lui, se situe ici dans notre espace de points. À votre avis, s'agit-il d'un spam ou non?



Si vous répondez *spam*, vous avez probablement raison. En effet, il semble plus logique de prédire que cet email soit un spam, car il est plus proche des points correspondant à cette classe.



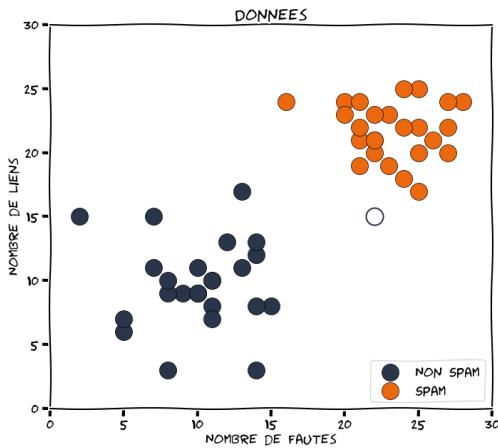
La méthode employée ici pour faire cette prédiction est celle des **plus proches voisins (nearest neighbors)**. Il s'agit d'une méthode très populaire en data science, particulièrement pour les problèmes de classification, et nous allons l'explorer en détail dans ce chapitre.

Bien sûr, il existe de nombreux autres modèles de classification en machine learning, parmi lesquels on retrouve :

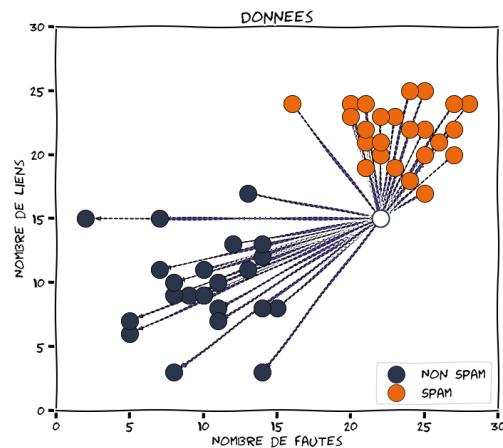
- La Régression Logistique
- Les Support Vector Machines
- Les Arbres de décision
- Les Random Forest
- ...

Modèle du Nearest Neighbors

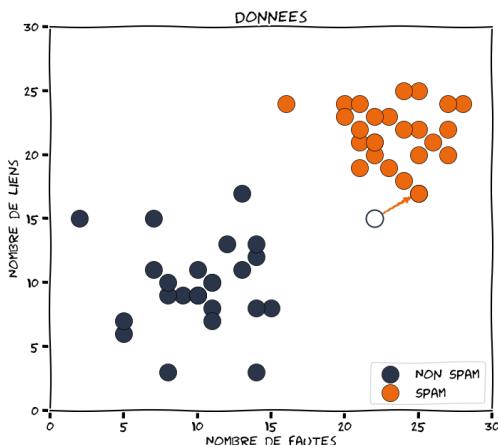
L'algorithme du *Nearest Neighbors* est particulier, car il nécessite de garder en mémoire tous les points du jeu de données pour calculer la distance entre ces points et un nouveau point dont on souhaite faire une prédiction.



Par exemple, voici un point dont on veut connaître la classe.

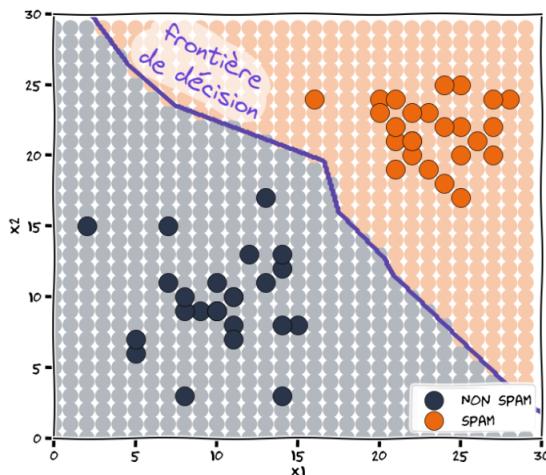
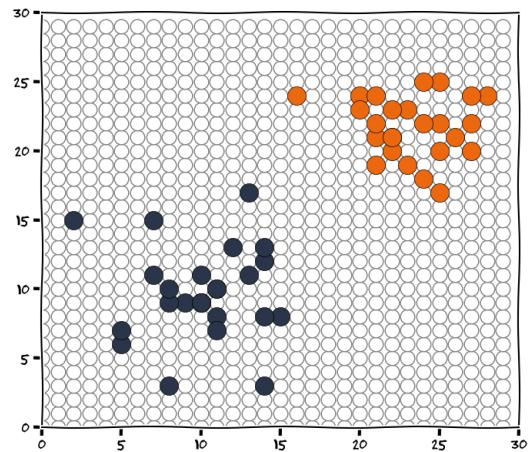


Pour cela, on calcule la distance entre ce point et tous les autres points du jeu de données. Il existe plusieurs méthodes pour calculer la distance entre deux points, mais bien souvent, on utilise la **distance euclidienne** (rappelez-vous, c'est la distance que nous avons vue dans le chapitre sur la régression)



En classant chaque distance de la plus petite à la plus grande, on identifie le point du jeu de données le plus proche. Dès lors, on attribue notre point à la même classe que son plus proche voisin.

Nous pouvons généraliser cette approche en appliquant cette opération à tous les points de notre espace...

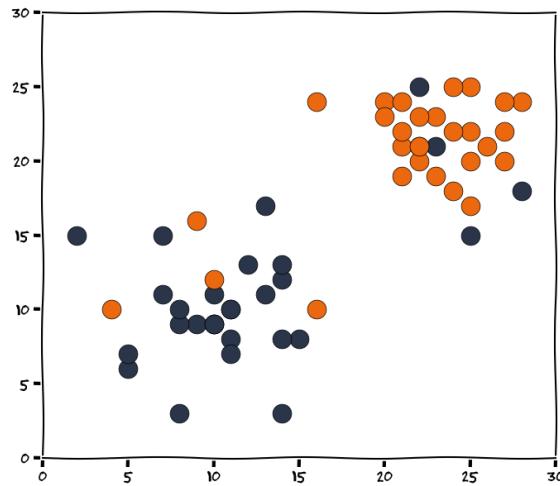


... Ce qui permet de mettre en avant la **frontière de décision**, c'est-à-dire la frontière qui sépare les deux classes.

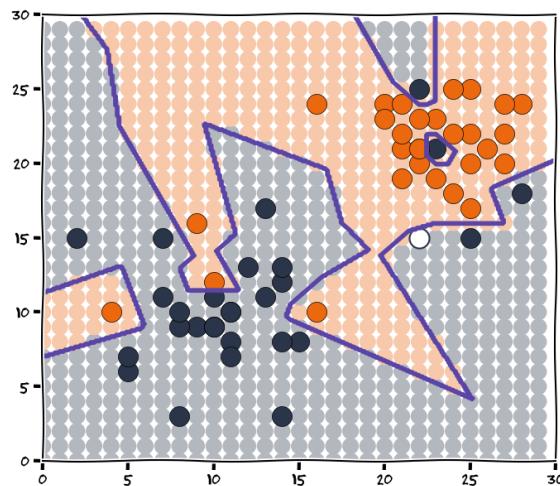
Et voilà, vous savez désormais comment fonctionne l'algorithme du plus proche voisin. Malgré sa simplicité, cet algorithme est encore largement utilisé aujourd'hui dans de nombreuses applications, y compris chez Spotify, Netflix, Amazon, et bien d'autres.

Généralisation : Le modèle du K-Nearest Neighbors

Dans sa forme la plus simple, l'algorithme du plus proche voisin, dans sa version à 1 voisin, n'est pas très robuste. En effet, voyons ce qu'il se produit lorsqu'on fait face à un jeu de données un peu plus complexe. Ci-dessous, quelques valeurs aberrantes se sont glissées dans chaque classes. Cependant, à vue d'œil, cela ne change pas vraiment la tendance globale qui suggère que les emails spams sont situés en haut à droite, et les non-spams en bas à gauche.

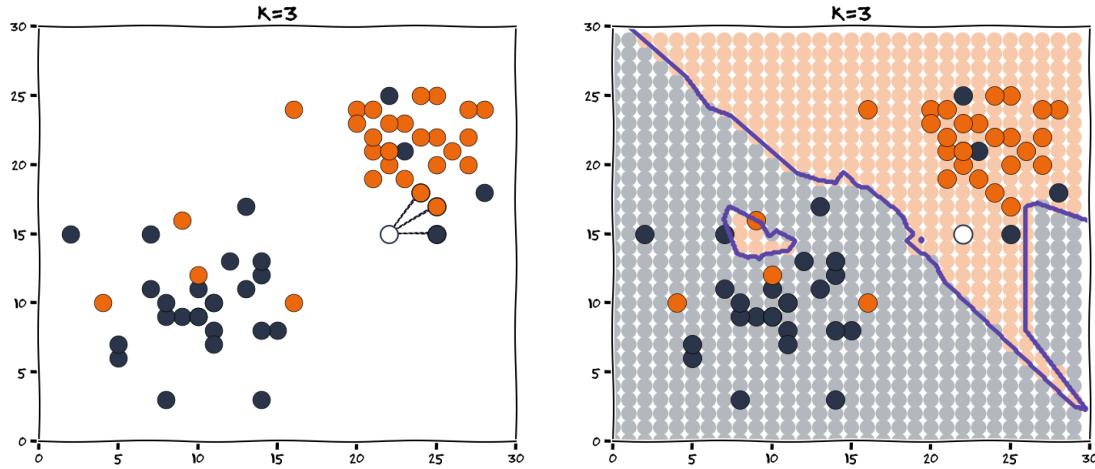


Malheureusement, la frontière de décision obtenue en suivant l'approche du plus proche voisin donne un résultat chaotique, qui ne correspond plus du tout à la tendance globale que nous avions observée précédemment. L'email de départ dont nous voulions connaître la classe est maintenant classé parmi les emails non-spams bien qu'il semble plus proche du groupe des spams.

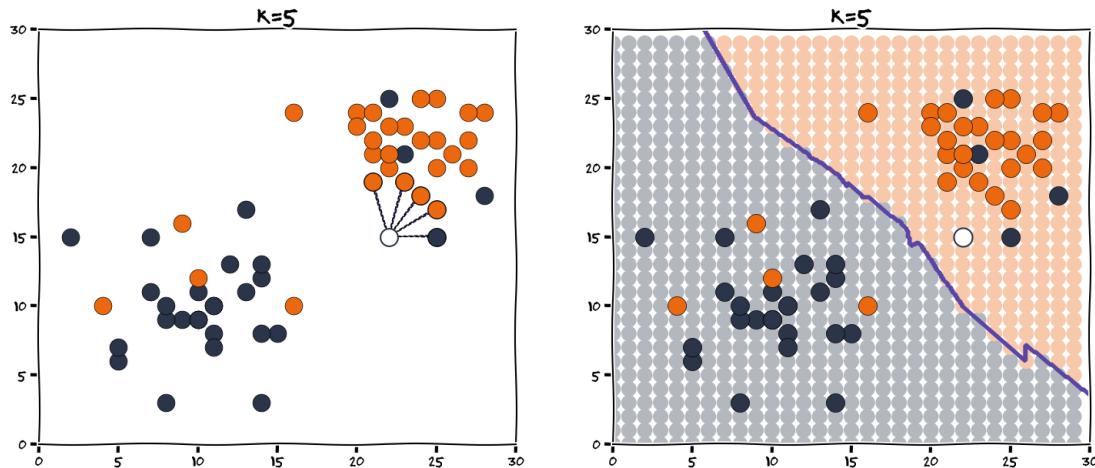


Pour éviter ce problème, on utilise généralement les k-voisins plus proches, pour ensuite assimiler notre point à la classe majoritaire parmi ces k-voisins.

Voici un exemple avec $k=3$. Ici, notre point compte parmi ses 3 voisins les plus proches, 2 points de la classe Orange et 1 point de la classe Gris foncé. Il est donc assimilé à la classe Orange. La frontière de décision est ainsi plus cohérente que précédemment.

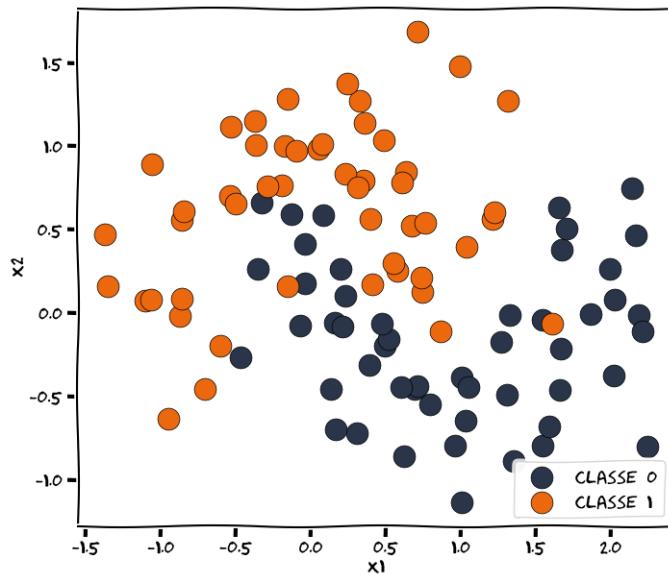


Et voici un exemple avec $K=5$.

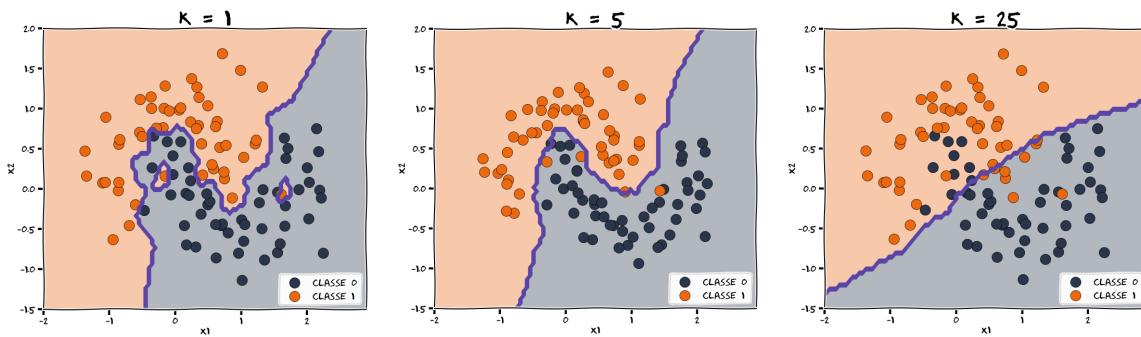


Comme on peut le constater, plus le nombre de voisins augmente, moins la frontière de décision est sensible aux valeurs aberrantes.

Cependant, il faut rester vigilant à ne pas trop augmenter le nombre de voisins, au risque de trop lisser la frontière de décision. L'exemple ci-dessous montre un jeu de données dans lequel les deux classes prennent une forme de croissant, avec une légère superposition de certains points.



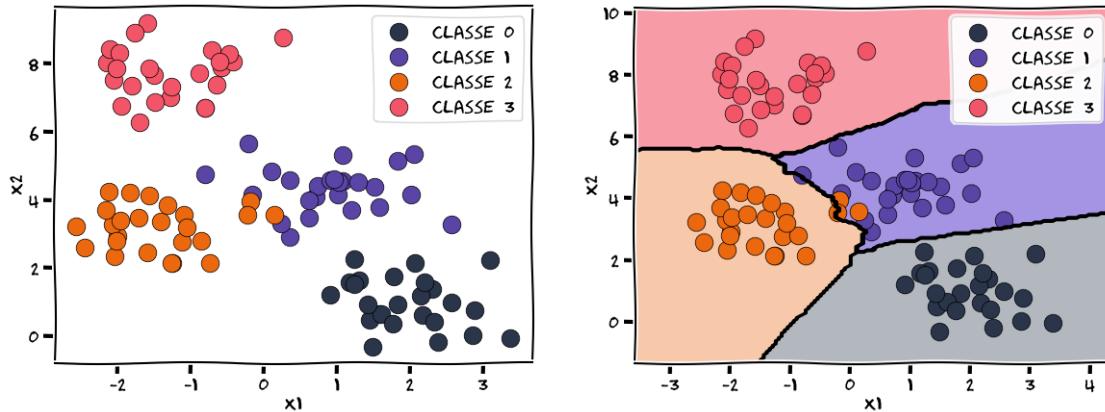
Ainsi, la situation $k = 1$ n'est pas idéale, car elle est trop sensible aux valeurs aberrantes, comme nous l'avons vu précédemment. Mais une valeur de k trop élevée à pour effet de tracer une frontière de décision qui ne correspond plus du tout à l'allure générale de notre jeu de données. Il convient donc de trouver le juste milieu.



Classification multiple

Les problèmes de classification ne se limitent pas à la classification binaire. En machine learning, il est possible de travailler avec autant de classes que l'on souhaite. Voici par exemple un jeu de données comprenant quatre classes.

L'algorithme des k-voisins les plus proches fonctionne parfaitement dans ce genre de situation, et nous n'avons rien à modifier dans notre algorithme pour obtenir ces résultats.



Implémentation Python

Voyons maintenant comment implémenter un algorithme des k plus proches voisins avec Python. Pour cela, nous avons besoin de la librairie de **scikit-learn** et de son module **neighbors**. Au sein de ce module, on retrouve le modèle **KNeighborsClassifier**.

Pour générer des données, nous allons également utiliser la fonction *make blobs* provenant de *scikit learn*.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.datasets import make_blobs
4 from sklearn.neighbors import KNeighborsClassifier

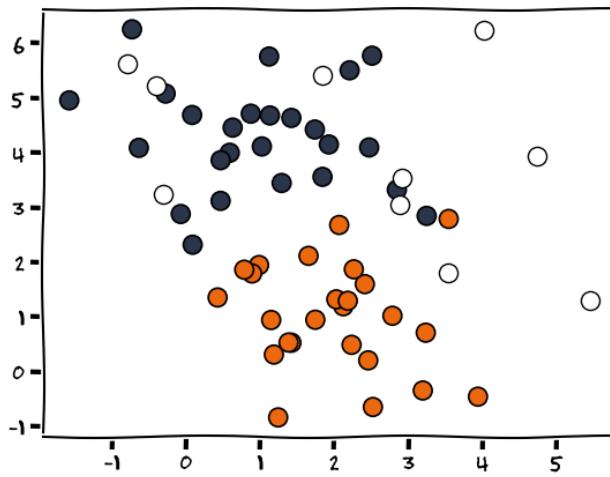
```

Commençons donc par simuler des données. Ici, nous allons créer deux classes y , avec 50 points, et deux variables X . Nous allons également générer des points dont la classe est inconnue.

```

1 X, y = make_blobs(n_samples=50, centers=2, n_features=2, random_state=0, cluster_std
2   ↪ =1)
3 X_new = np.random.uniform(low=X.min(), high=X.max(), size=(10,2))
4 plt.scatter(X[:, 0], X[:, 1], c=y, cmap="bwr")
5 plt.scatter(X_new[:, 0], X_new[:, 1], c='white', edgecolor='k')
6 plt.show()

```

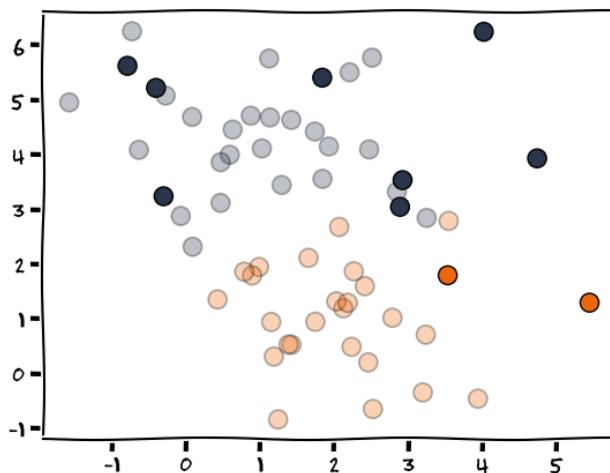


Pour entraîner le modèle de KNeighborsClassifier, nous allons l'initialiser avec une valeur k=3, puis utiliser la méthode fit() comme vu dans le chapitre sur la régression.

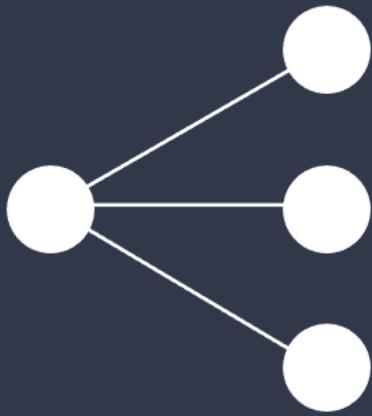
```
1 model = KNeighborsClassifier(n_neighbors=3)
2 model.fit(X, y)
```

Enfin, pour effectuer des prédictions et classer les nouveaux points de notre jeu de données, on utilise la méthode predict(). On arrive ainsi à classer les nouveaux points du jeu de données.

```
1 predictions = model.predict(X_new)
2
3 plt.scatter(X[:, 0], X[:, 1], c=y, alpha=0.4)
4 plt.scatter(X_new[:, 0], X_new[:, 1], c=predictions, edgecolor='k')
5 plt.show()
```



Clustering



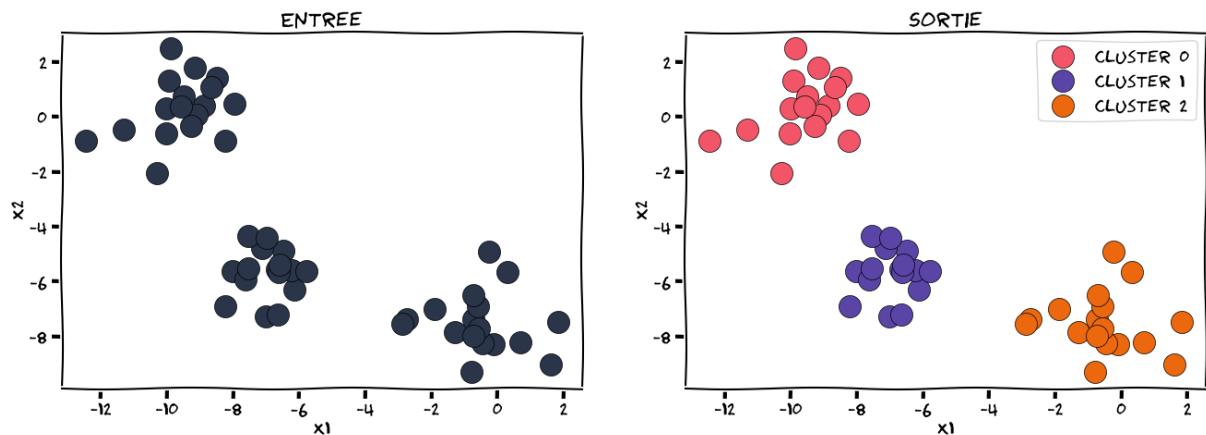
Clustering

Dans ce chapitre, vous allez apprendre à développer des modèles de **clustering**. En machine learning, ce type de modèle est utilisé lorsque l'on souhaite classer nos données de manière non supervisée. Contrairement à ce que nous avons vu dans le chapitre précédent, le clustering consiste donc à regrouper nos données en nous basant uniquement sur leur ressemblance, sans se préoccuper d'une quelconque classe "correcte" ou "incorrecte".

Par exemple :

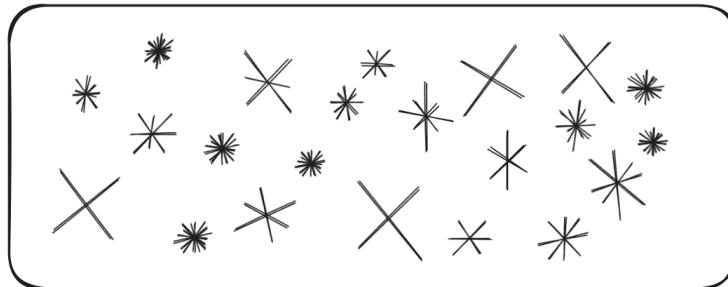
- Classer des images ou des objets selon leur ressemblance
- Segmenter une base de données de clients, selon leurs habitudes de consommation
- Regrouper des documents selon leur contenu.
- etc.

Le clustering est donc une méthode intéressante lorsqu'on souhaite laisser à la machine le pouvoir de proposer sa propre solution, et ainsi découvrir une approche différente de la nôtre.

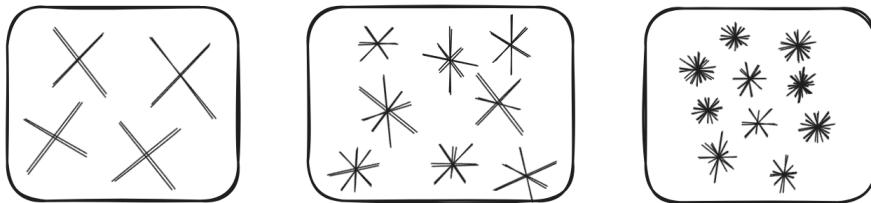


Mise en situation

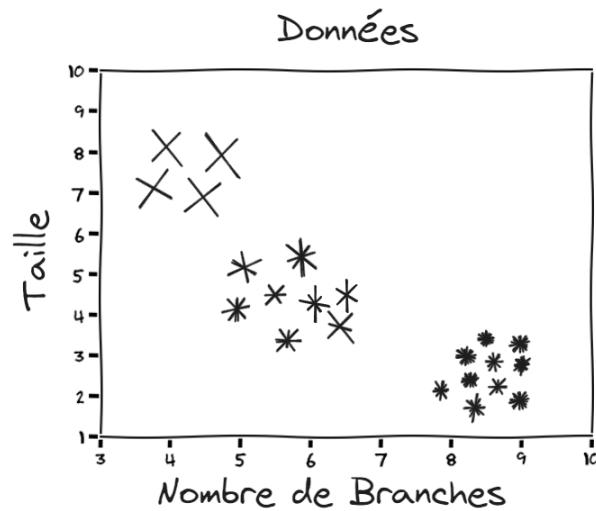
Observez attentivement les formes ci-dessous. Elles ressemblent peut-être à des étoiles, mais ne représentent en réalité rien de concret. Vous pouvez donc laisser libre cours à votre imagination en les assimilant à des objets, des biens immobiliers, des individus, etc.



À présent, votre tâche consiste à regrouper ces formes selon leur ressemblance, afin de créer trois groupes distincts. La grande majorité des gens formeraient les trois groupes suivants :



Les caractéristiques qui vous ont probablement permis de trier ces objets sont la taille et le nombre de branches de chaque forme. Ainsi, en mesurant ces attributs pour chaque objet et en les plaçant sur un graphique, on obtient le résultat suivant :



Sur ce graphique, on distingue très facilement la présence de trois groupes. Est-il alors nécessaire de superviser l'apprentissage de la machine en désignant des classes y pour que celle-ci puisse classer ces points ? La réponse est non.

Le principe du clustering est d'analyser les différentes variables X qui caractérisent nos données, afin de regrouper les points en clusters, sans pour autant connaître la nature de ces points.

De nombreux algorithmes permettent de réaliser du clustering :

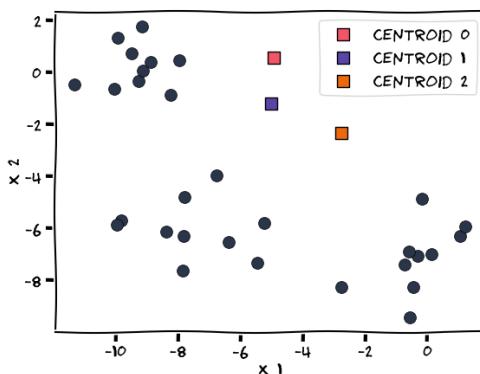
- Le K-Means Clustering
- Le Clustering hierarchique
- DBSCAN
- OPTICS
- etc.

Dans ce chapitre, nous allons voir l'algorithme de K-Means Clustering, de loin le plus populaire et simple à comprendre.

Fonctionnement du modèle de K-Means Clustering

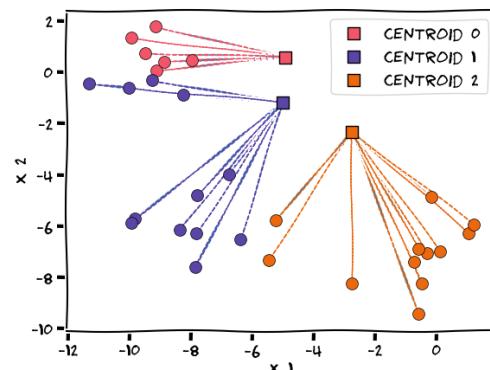
Le K-Means Clustering est un algorithme itératif, dans lequel on demande à notre machine de trouver un nombre K de clusters au sein de notre jeu de données. Pour cela, elle place au hasard K points dans l'espace, puis déplace ces points pour qu'ils deviennent les barycentres de nos clusters.

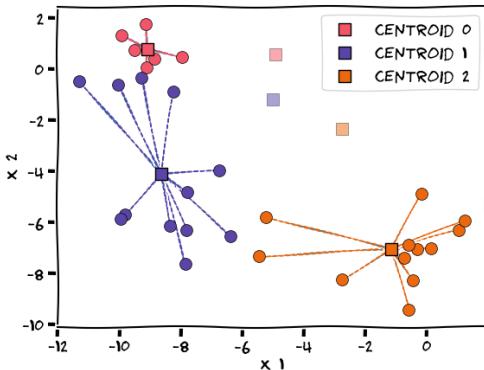
Prenons un exemple avec $K=3$ clusters.



Pour commencer, on place trois points aléatoires au sein de nos données. Ces points sont appelés centroïdes et sont les futurs centres de masse des clusters que l'on veut former.

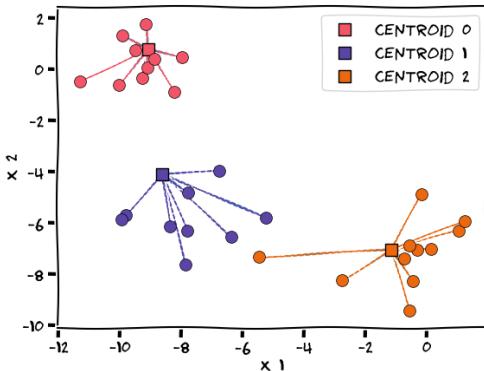
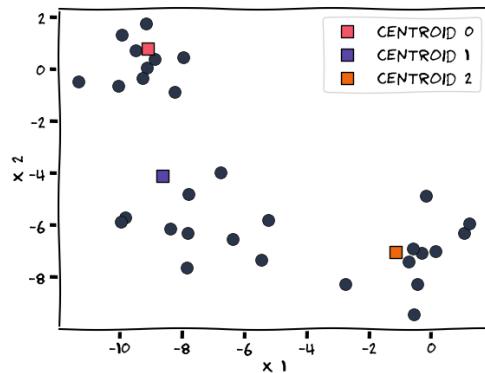
Ensuite, on associe chaque point de notre jeu de données au centroïde dont il est le plus proche, en calculant la distance euclidienne (comme nous l'avons fait dans le chapitre 3)





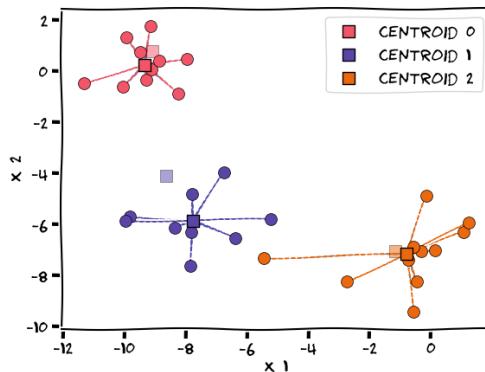
Ensuite, on déplace chaque centroïde au centre de son groupe, ce qui explique son nom de "centroïde". Pour ce faire, on calcule la moyenne des points du cluster, et cette moyenne depuis la nouvelle position du centroïde, d'où le nom de "K-Means".

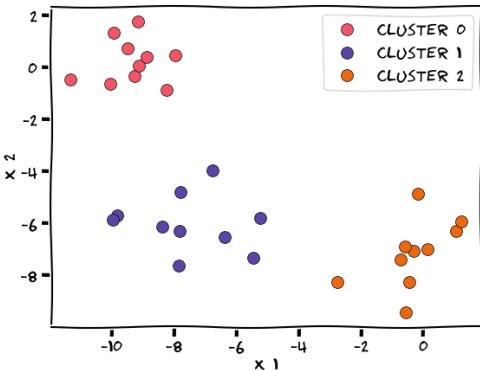
Comme la position de chaque centroïde a changé, les centroïdes ont désormais des plus proches voisins différents. Ainsi, nous réitérons notre algorithme...



... Nous réassignons chaque point à son centroïde le plus proche (notez bien que cela produit des résultats différents de ceux de la première itération!).

... Puis on déplace chaque centroïde au centre de son groupe.



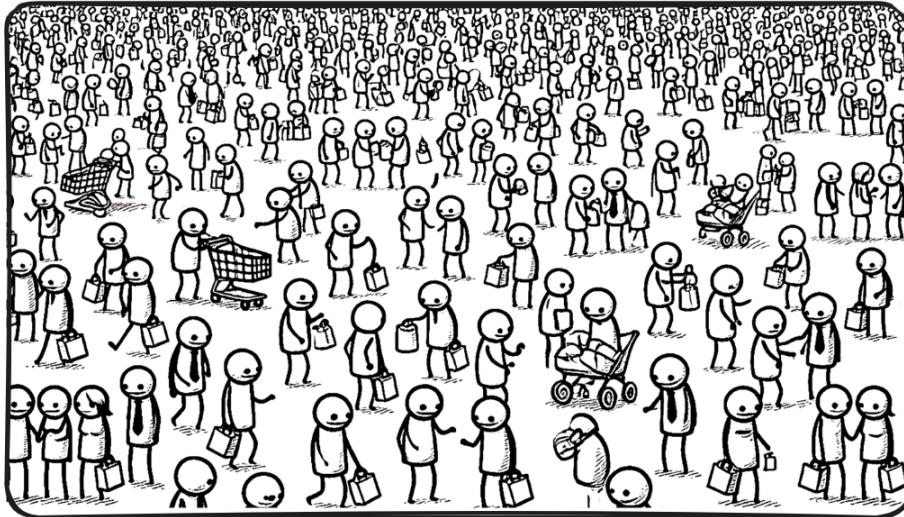


Et cet algorithme se répète ainsi jusqu'à ce que les centroïdes ne bougent plus. C'est alors que l'algorithme se termine.

Et voilà, c'est ainsi que fonctionne l'algorithme du K-Means clustering. Il vous suffit de déclarer le nombre de clusters que vous souhaitez obtenir, et l'algorithme s'occupe du reste.

Mais comment déterminer le nombre de centroïdes à utiliser dans un projet? Dans le cas présent, la réponse était relativement simple, car nous pouvions observer l'allure du jeu de données, et discerner visuellement trois groupes de points avant même l'entraînement du modèle.

Dans la pratique, les choses sont souvent plus complexes. Imaginez devoir segmenter une bases de données clients, comprenant des centaines de variables X , tels que leurs habitudes de consommation, leur âge, leur localisation, leur genre, etc.



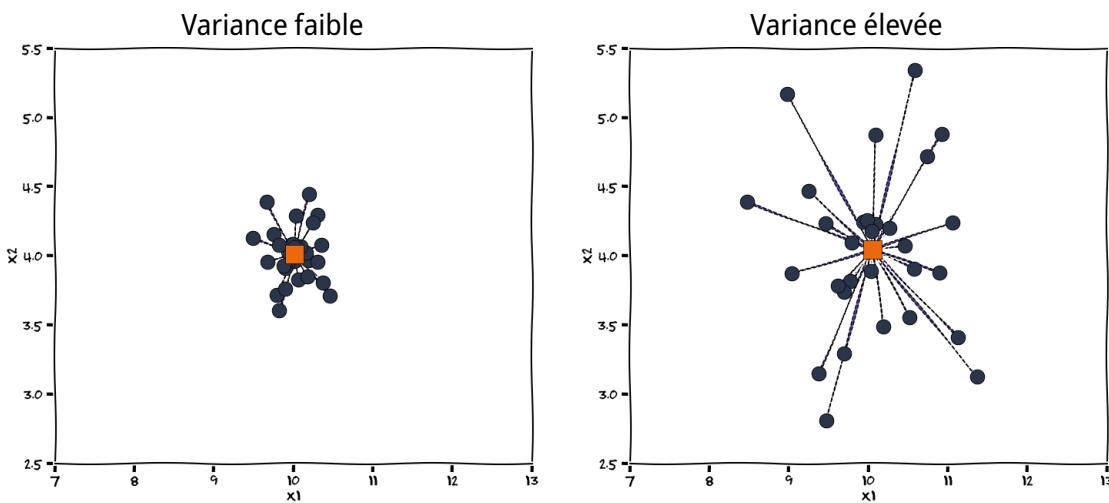
Dans ce cas, il n'est pas possible de visualiser les données pour déterminer le nombre optimal de clusters.

Heureusement, il existe une méthode permettant de trouver le nombre optimal de clusters sans avoir à observer directement notre jeu de données. Cette méthode s'appelle la méthode du Coude.

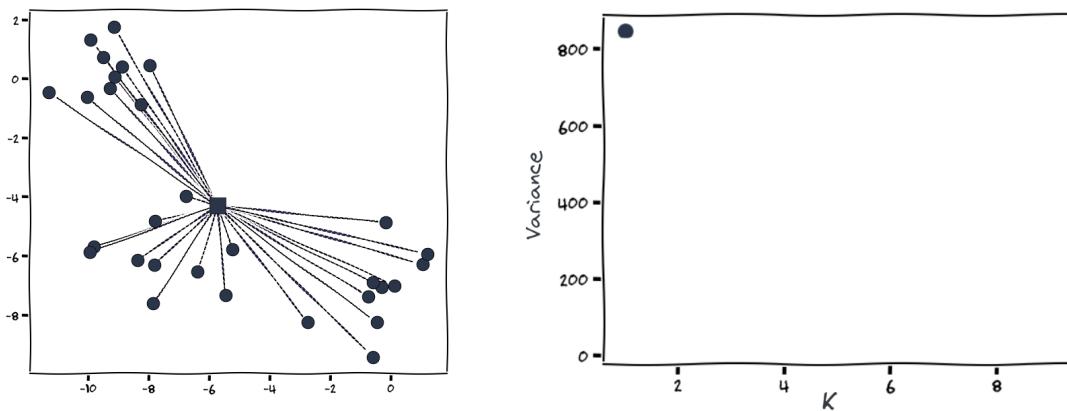
La méthode du coude

La méthode du coude consiste à tester différentes valeurs de K pour l'algorithme du K-Means Clustering, en mesurant à chaque fois la variance finale de chaque clustering, puis en traçant sur un graphique l'évolution de cette variance en fonction de K pour identifier une forme de coude dans le graphique. Laissez-moi vous expliquer...

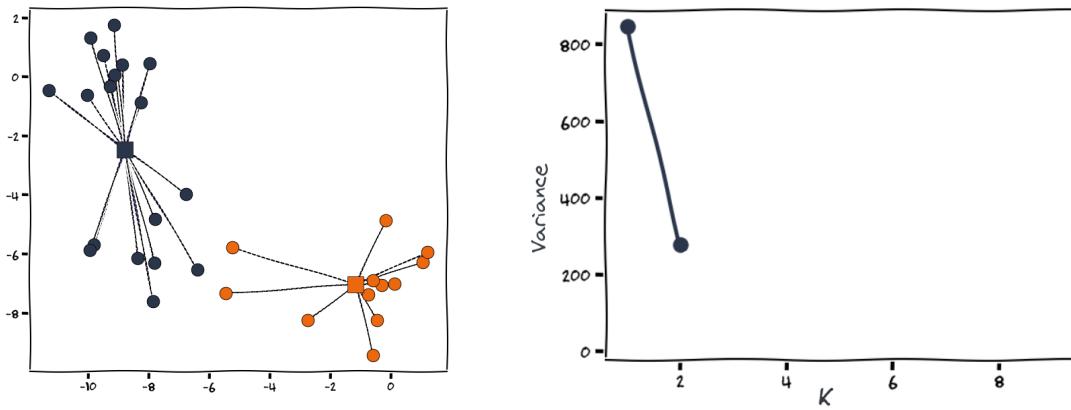
Commençons par nous rappeler ce qu'est la **variance**. Il s'agit de l'écart des points par rapport à leur moyenne, autrement dit, la distance globale des points d'un cluster par rapport à leur centroïde.



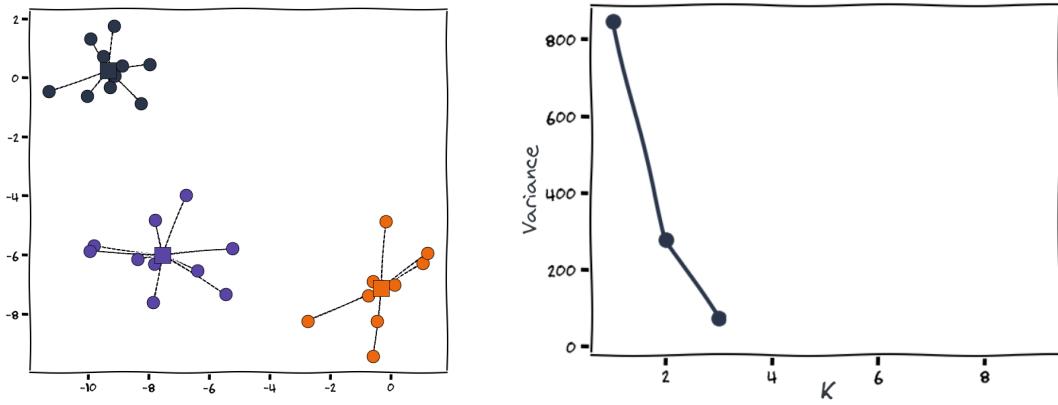
Maintenant que nous nous rappelons de ce qu'est la variance, reprenons notre jeu de données et voyons ce qu'il se passe lorsque nous fixons K=1. On observe une grande distance entre les points et leur centroïde, ce qui signifie que la variance est donc élevée. Enregistrons cette valeur de variance sur le graphique de droite, sous la valeur K=1.



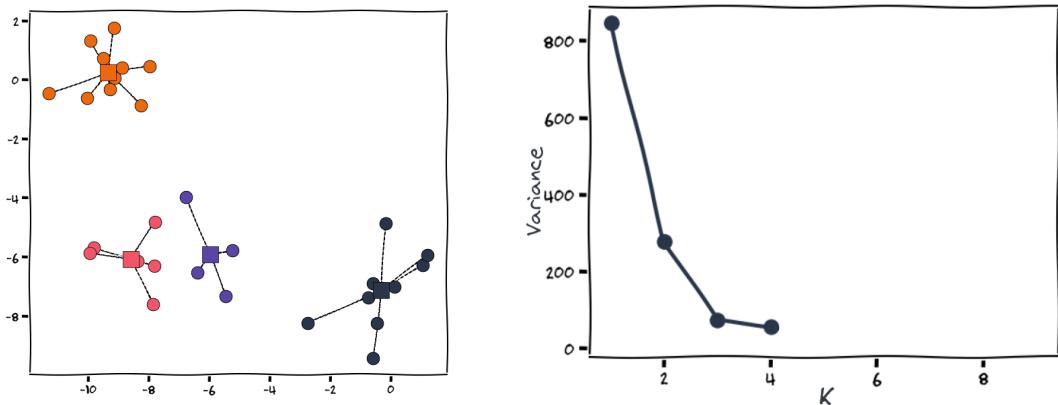
En passant à K=2 clusters, la distance moyenne entre les points et leur centroïde respectif diminue par rapport au cas précédent. Nous ajoutons cette nouvelle valeur à notre graphique précédent.



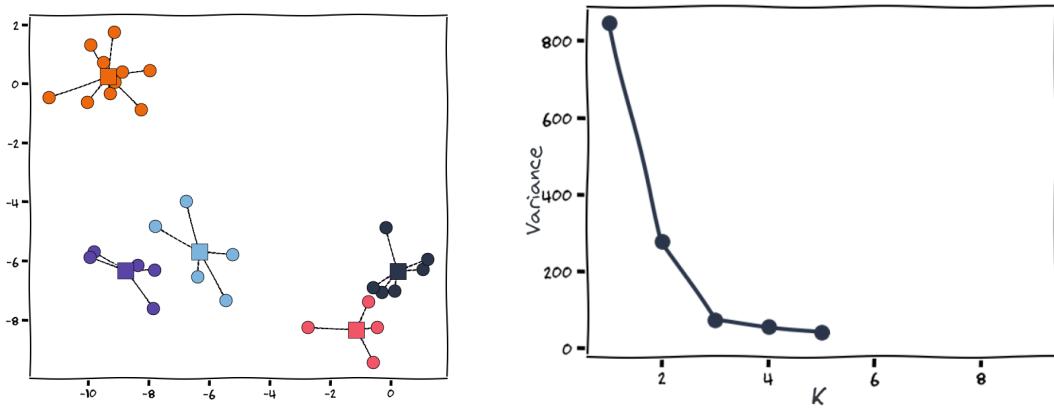
En poursuivant avec $K=3$ clusters, la variance continue de diminuer...



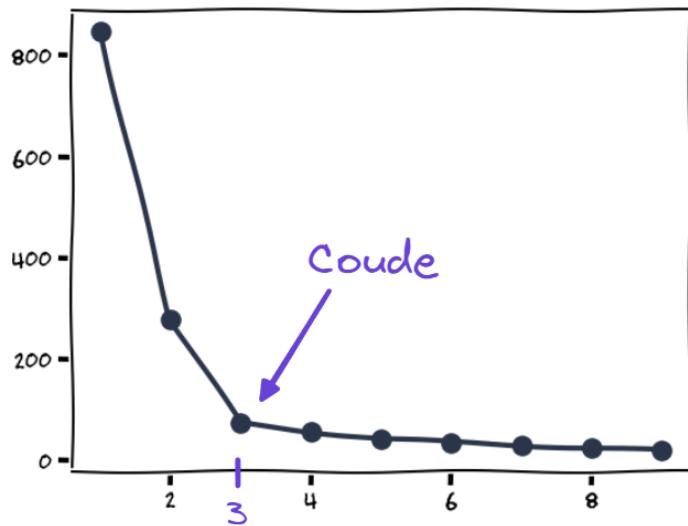
C'est à partir de maintenant que les choses changent. Pour $K=4$ clusters, la variance n'est pas très différente de celle pour $K=3$. Cela nous indique que nous commençons à diviser un cluster qui avait déjà une variance assez faible. Nous entrons donc dans un phénomène de sur-optimisation, ce que nous souhaitons éviter.



On continue de voir ce phénomène pour $K=5$...



En continuant ainsi jusqu'à $K=9$, on observe sur notre graphique l'apparition d'une forme de coude. C'est à ce point que K semble être optimal, ce qui est cohérent avec nos données.



Dans la pratique, les choses ne sont pas aussi évidentes qu'ici, mais la technique du coude (*elbow method* en anglais) est très populaire en clustering pour découvrir comment segmenter un jeu de données complexe et riche. Je vous encourage à l'utiliser dans vos projets.

Voyons maintenant comment implémenter tout cela en Python !

Implémentation en Python

Pour développer un modèle de K-Means clustering, nous allons une fois de plus utiliser scikit-learn, en chargeant le modèle **KMeans** depuis le module *cluster*.

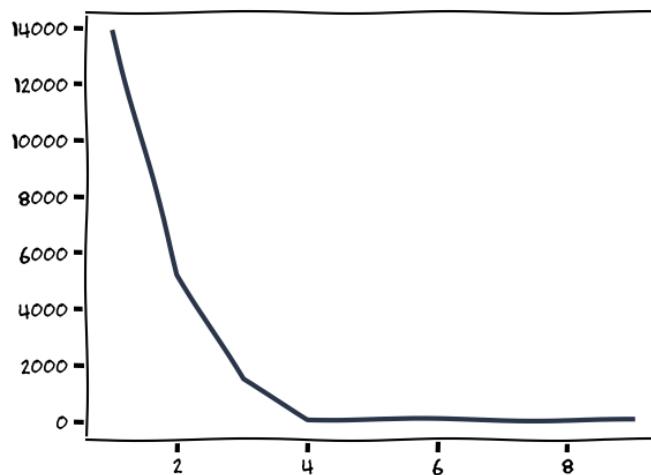
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.datasets import make_blobs
4 from sklearn.cluster import KMeans
```

Pour vous démontrer l'efficacité de la méthode du coude, nous allons générer un jeu de données avec un nombre aléatoire de clusters. Ce nombre nous sera inconnu, et nous devrons le découvrir à l'aide de la méthode du coude.

```
1 k = np.random.choice([2, 3, 4, 5, 6, 7, 8])
2 X, _ = make_blobs(n_samples=200, centers=k, n_features=2, cluster_std=0.5)
```

Pour programmer cette méthode, nous allons utiliser une boucle *for*, qui nous permettra d'entraîner un modèle de KMeans pour différentes valeurs de K allant de 1 à 9.

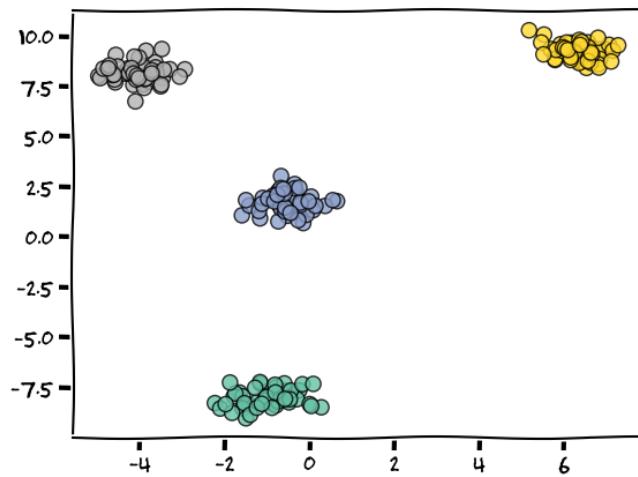
```
1 k_range = range(1, 10)
2 k_inertias = []
3
4 for k in k_range:
5     model = KMeans(n_clusters=k).fit(X)
6     k_inertias.append(model.inertia_)
7
8 plt.plot(k_range, k_inertias, lw=3)
9 plt.show()
```



Le graphique résultant de cette expérience semble montrer un coude pour la valeur de K = 4
(Note : Parfois, la position du coude peut être difficile à identifier...)

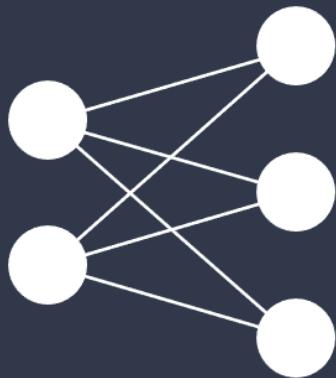
Ainsi, nous allons créer un modèle KMeans avec un nombre de clusters égal à 4, l'entraîner sur les données X et observer les résultats.

```
1 model = KMeans(n_clusters=4)
2 model.fit(X)
3 predictions = model.predict(X)
4
5 plt.scatter(X[:, 0], X[:, 1], c=predictions)
6 plt.show()
```



Bingo! Il semblerait que notre analyse était juste, on constate que notre jeu de données comprenait bien 4 clusters, et nous avons pu les identifier grâce à la méthode du coude, tout en réalisant correctement le travail de clustering.

Détection d'anomalies



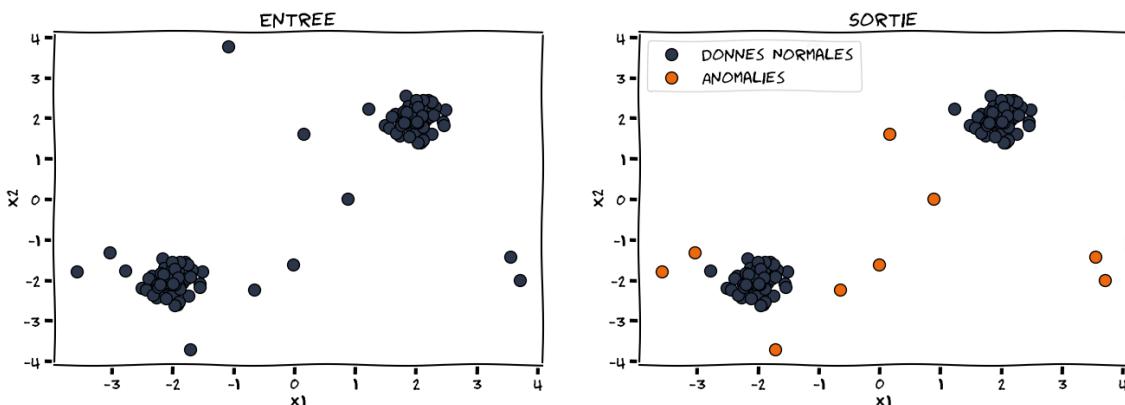
Détection d'anomalies

Dans ce chapitre, vous allez apprendre à développer des modèles de détection d'anomalie. En machine learning, ces modèles sont utilisés pour identifier, de manière non supervisée, tout point s'écartant trop des autres points d'un jeu de données.

L'idée repose sur le fait qu'une anomalie est, par définition, un évènement rare (sinon, cela devient une "norme"). Ses caractéristiques X sont donc très éloignées de la majorité des autres points.

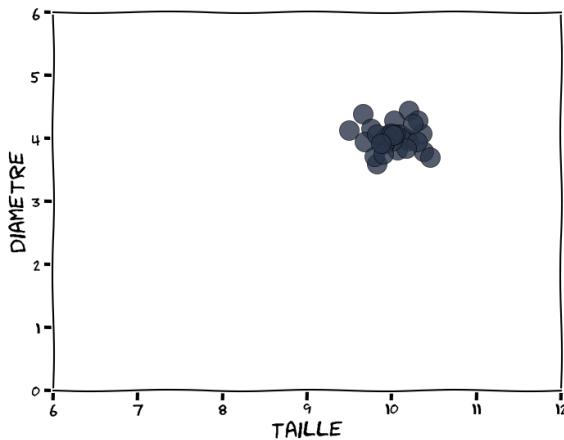
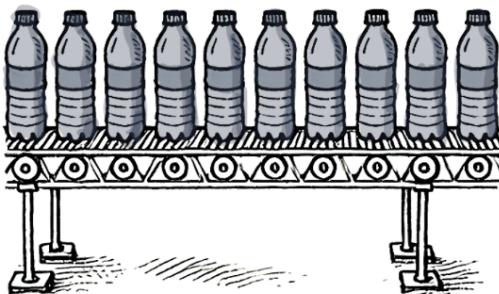
Par exemple :

- Une bouteille d'eau avec une forme très différente des autres sur une chaîne de production
- Une transaction bancaire frauduleuse, qui ne ressemble en rien aux transaction habituelles de votre compte
- Un piéton qui marche dans une zone où personne ne marche habituellement

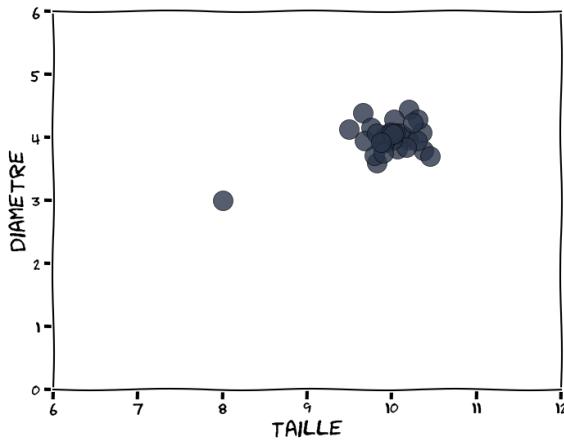
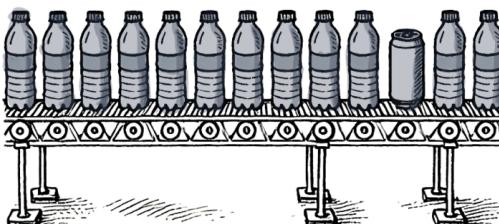


Mise en situation

Prenons l'exemple d'une chaîne de fabrication de bouteilles d'eau. La grande majorité des bouteilles ont une forme très semblable les unes aux autres. Si l'on affiche dans un graphique la taille des bouteilles x_1 et leur diamètre x_2 , nous observons le nuage de points suivant.

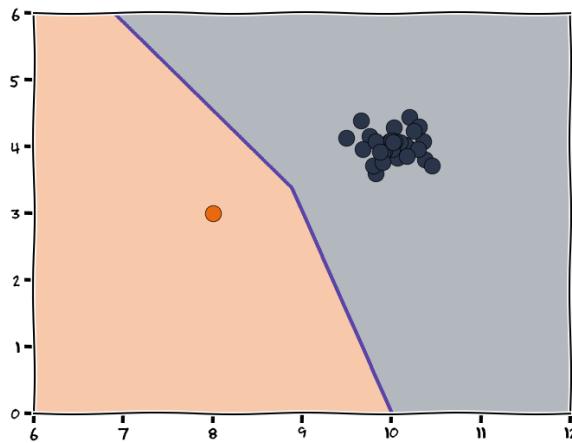


Maintenant, une bouteille très différente des autres passe devant nos capteurs..

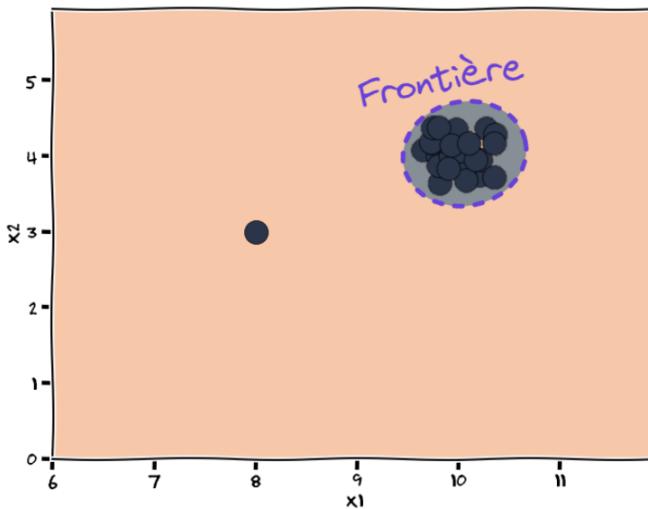


Inutile de préciser où se situe ce point dans les données, vous l'aurez trouvé vous-même. Cela prouve que ce genre de problème peut être résolu de manière non supervisée, sans avoir besoin de fournir d'informations sur une quelconque classe *y* "normale" / "anomalie".

En réalité, il pourrait même être contre-productif d'aborder un problème de détection d'anomalies avec une approche supervisée, comme la méthode de classification vue dans le chapitre 3. En effet, on obtiendrait une frontière de décision qui séparerait les données comme illustré ci-dessous.



À la place, nous cherchons à développer un modèle qui puisse déclarer comme anomalie tout point déviant trop de la norme. Ainsi, notre frontière de décision devrait plutôt ressembler à ceci :



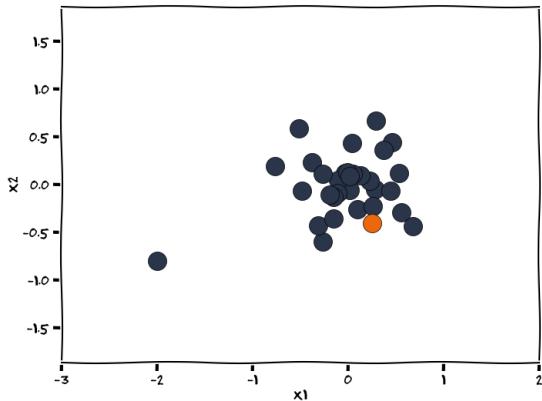
Voici quelques exemples d'algorithmes permettant de résoudre ce genre de problème en machine learning :

- Local Outlier Factor
- Isolation Forest (que j'aborde sur la chaîne YouTube Machine Learnia)
- Auto-Encodeur

Dans ce chapitre, nous allons démystifier le modèle du **Local Outlier Factor** (LOF).

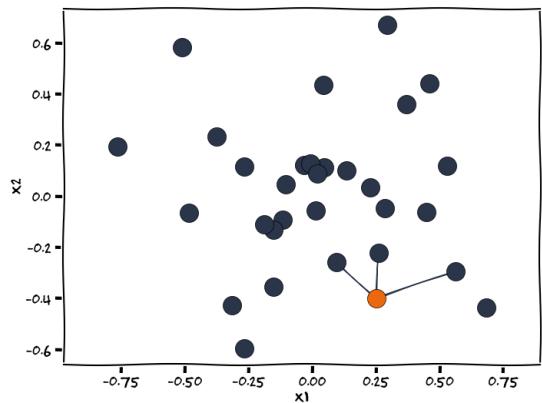
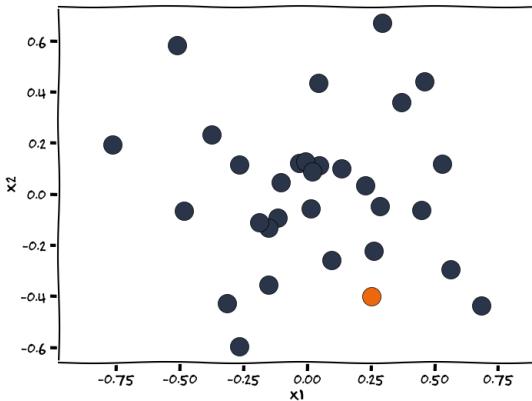
Local Outlier Factor

L'algorithme du Local Outlier Factor (LOF) consiste à calculer l'écart de densité locale d'un point par rapport à celle de ses voisins. Il considère comme anomalies les points ayant une densité nettement inférieure à celle de l'ensemble des autres points.



Prenons un point au hasard dans notre jeu de données. Ici en orange.

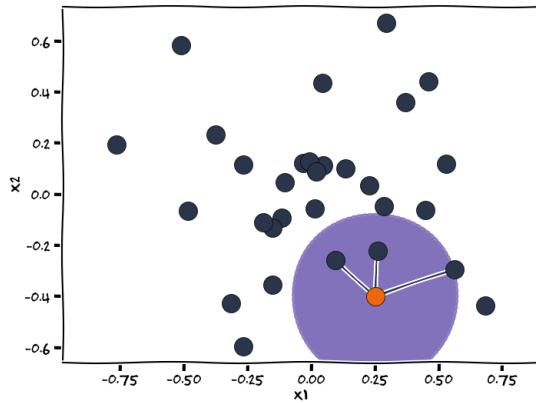
Rapprochons nous un peu plus de ce point pour y voir plus clair.



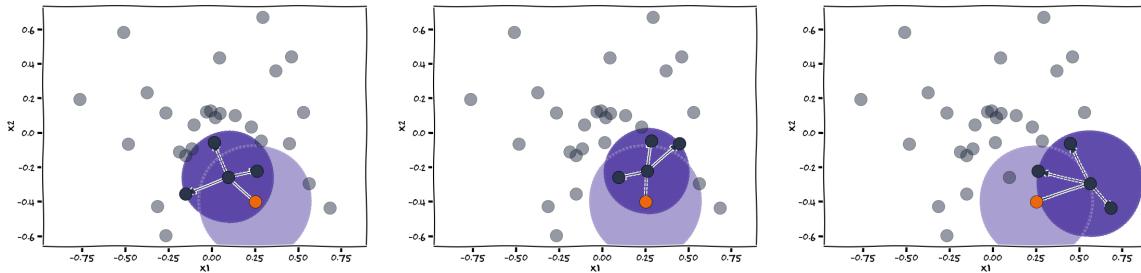
L'algorithme du LOF commence par trouver les K-voisins les plus proches de ce point.

Pour cet exemple, posons $K = 3$.

Ensuite on trace un rayon d'action basé sur la distance au voisin le plus éloigné. C'est ce que l'on appelle la **distance-k**.



Puis, pour chacun des k-voisins du point orange, nous procédons de la même manière : nous cherchons les k-voisins les plus proches, et traçons la distance-k sous forme de cercle.

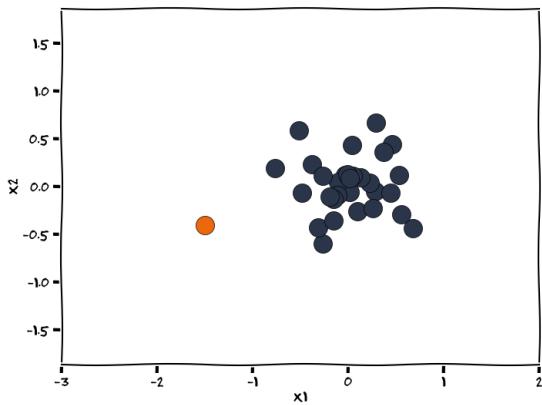


Pour finir, notre algorithme calcule le rapport entre la distance-k du point orange et la moyenne des distances-k de ses plus proches voisins (en réalité, c'est un peu plus complexe, mais l'idée centrale est là). Le nombre obtenu est le facteur local d'anomalie (*Local Outlier Factor*). Plus ce facteur est élevé, plus il est probable que notre point soit considéré comme une anomalie.

Ici, les k-distances sont à peu près équivalentes, et leur rapport est donc d'environ 1, ce qui indique que ce point n'est probablement pas une anomalie.

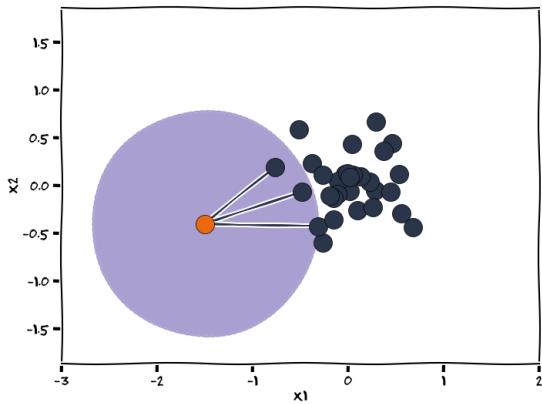
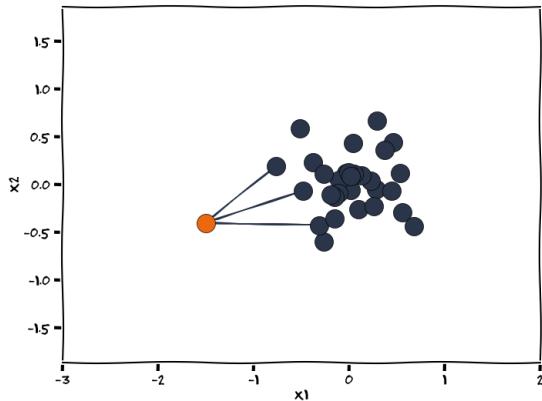
$$\text{LOF} = \frac{\text{Distance-k of point of interest}}{\frac{\text{Distance-k of point of interest} + \text{Distance-k of its 1st neighbor} + \text{Distance-k of its 2nd neighbor}}{3}} = \frac{\text{Distance-k of point of interest}}{\frac{\text{Distance-k of point of interest} + \text{Distance-k of point of interest}}{2}} = 1$$

The diagram illustrates the calculation of the Local Outlier Factor (LOF). It shows a point of interest (orange dot) with a purple circle representing its k-distance neighborhood. Below it, three smaller purple circles represent the k-distance neighborhoods of its three nearest neighbors. The formula calculates the ratio of the point's own k-distance to the average of its neighbors' k-distances. In this case, all k-distances are approximately equal, resulting in a LOF of 1, which indicates the point is not an outlier.



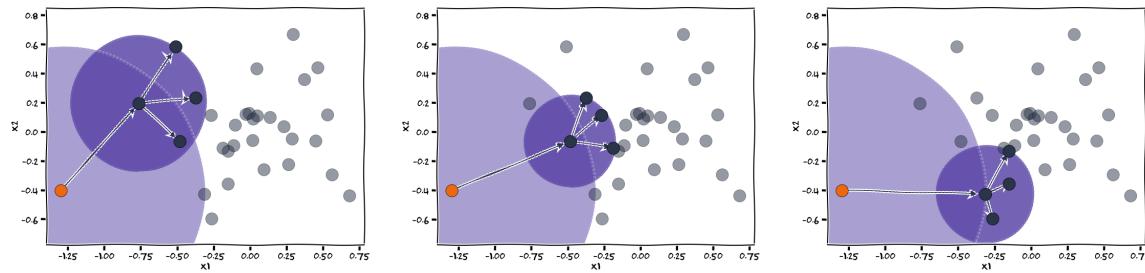
Prenons maintenant un point clairement éloigné de la masse. À première vue, nous savons qu'il s'agit d'une anomalie, mais cela, notre machine ne le sait pas encore.

Voici ses k-voisins les plus proches..

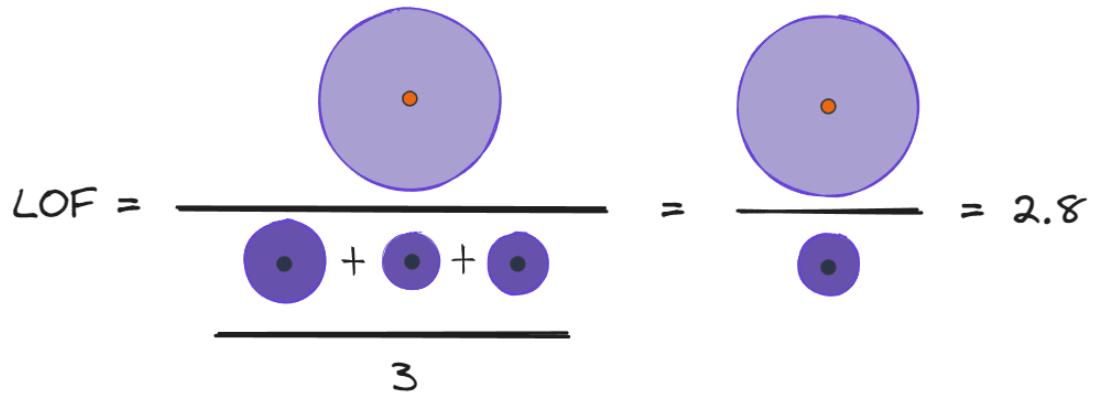


... Ainsi que sa distance- k (le rayon d'action lié au voisin le plus éloigné)

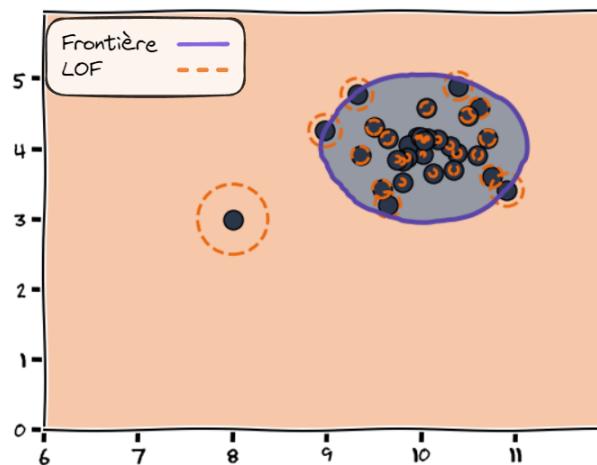
Pour chacun de ses voisins, nous effectuons la même opération qu'auparavant, à savoir : trouver ses k-voisins les plus proches, et calculer sa distance- k



Cette fois-ci, le rapport des distances-k donne un résultat bien plus élevé que précédemment. Cela signifie que le point orange dont nous mesurons le facteur d'anomalie, est beaucoup plus éloigné de ses voisins que ses voisins ne le sont les uns des autres.



En effectuant ce calcul pour tous les points du jeu de données, nous sommes capable de détecter ceux dont le LOF est plus élevé que la moyenne, et ainsi de les marquer comme des anomalies. Voici à quoi ressemble la frontière de décision ainsi obtenue :



Implémentation Python

Pour implémenter l'algorithme de Local Outlier Factor, nous allons importer la classe du même nom depuis le module *neighbors* de scikit-learn (le même module que celui utilisé dans le chapitre 3). À ce stade du livre, vous devriez avoir réalisé que de nombreux algorithmes de machine learning reposent sur le concept de mesure des distances entre points. Je trouve cette simplicité particulièrement élégante. Le machine learning, c'est vraiment cool! 😊

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.datasets import make_blobs
4 from sklearn.neighbors import LocalOutlierFactor

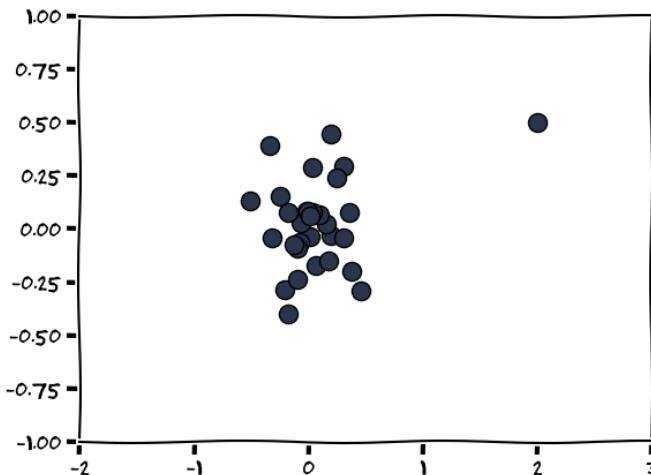
```

Commençons par générer quelques points au hasard, ainsi qu'une valeur aberrante isolée dans le jeu de données.

```

1 np.random.seed(0)
2
3 X = np.random.randn(30, 2) * 0.2
4 X = np.concatenate((X, np.array([[2, 0.5]])), axis=0)
5
6 plt.scatter(X[:, 0], X[:, 1])
7 plt.xlim(-2, 3)
8 plt.ylim(-1, 1)
9 plt.show()

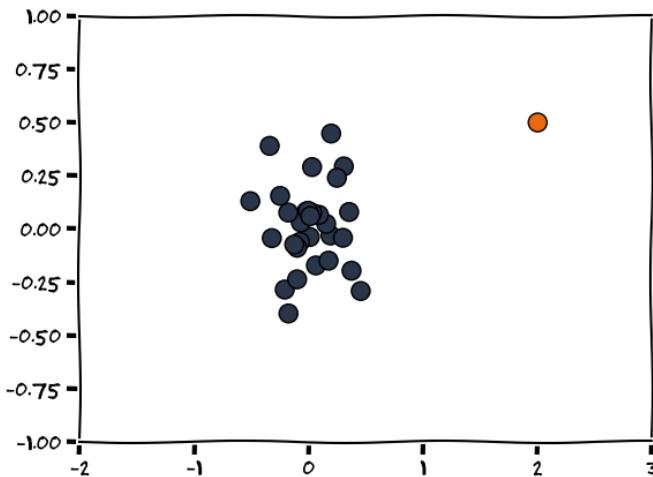
```



Nous allons ensuite créer un modèle de LoF, en spécifiant le nombre de voisins que nous souhaitons prendre en compte. Plus ce nombre est élevé, plus l'algorithme mettra en avant les points isolés du groupe.

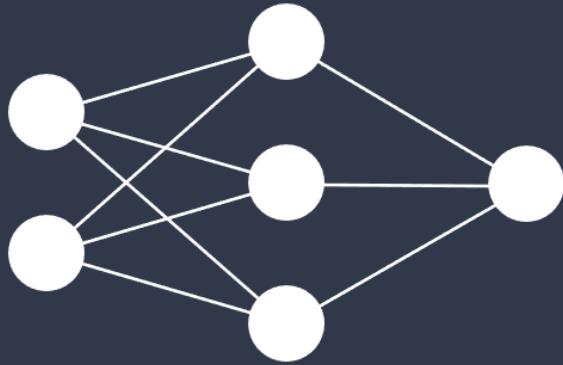
```
1 model = LocalOutlierFactor(n_neighbors=20, contamination=0.5)
2 predictions = model.fit_predict(X)
```

```
1 plt.scatter(X[:, 0], X[:, 1], c=predictions, cmap="bwr")
2 plt.xlim(-2, 3)
3 plt.ylim(-1, 1)
4 plt.show()
```



Si cet algorithme vous a plu, je vous donne le lien vers l'article scientifique à l'origine de cet algorithme [ici](#). Il est un peu complexe, c'est pourquoi je l'ai vulgarisé pour vous dans ce livre. ☺

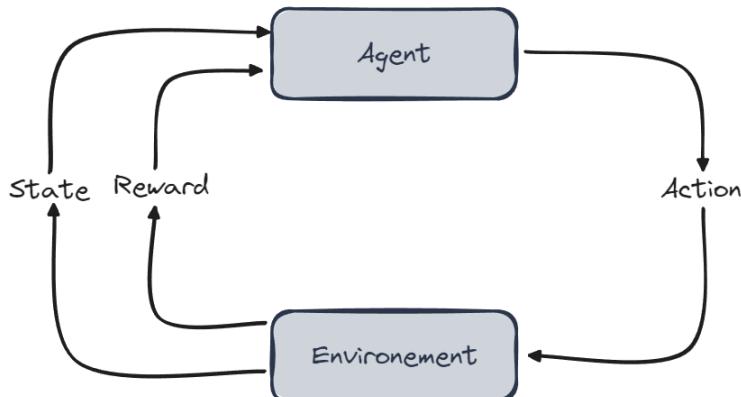
Apprentissage par renforcement



Apprentissage par renforcement

Dans ce chapitre, vous allez apprendre à développer des modèles d'apprentissage par renforcement. En machine learning, l'apprentissage par renforcement est une branche très différente de celles de l'apprentissage supervisé et non supervisé. En effet, dans cette discipline, on ne fournit pas de données (X, y) à notre machine, mais on la programme afin qu'elle puisse générer ses propres expériences et apprendre de ses erreurs.

L'idée centrale est donc de créer un **agent**, libre d'entreprendre des **actions** dans un **environnement**. Ces actions modifient l'**état** (*state*) de l'agent, et ce changement d'état s'accompagne d'une **récompense** (*reward*). Pour l'agent, le but du jeu est de maximiser ses récompenses, ce qui le pousse à apprendre quelles actions effectuer pour obtenir le plus de récompenses.



Cette discipline est très utilisée pour les applications suivantes :

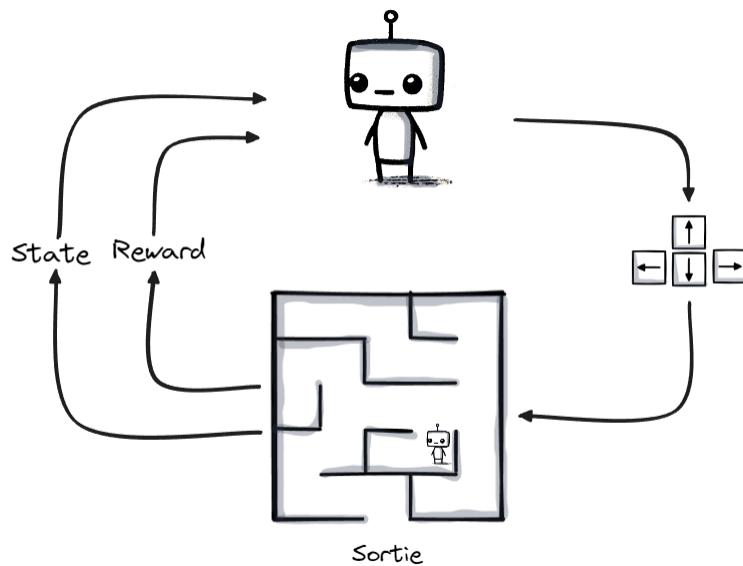
- Robotique
- Véhicules autonomes (voitures, drones)
- Trading
- Algorithmes de prise de décision

Mise en situation

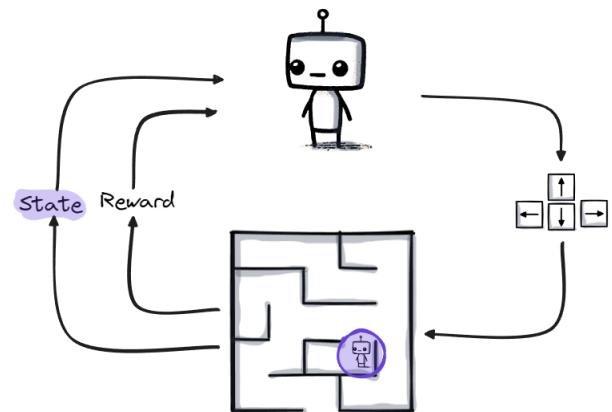
Imaginez : vous programmez un petit robot pour qu'il apprenne à sortir d'un labyrinthe le plus rapidement possible. Pour rendre les choses plus compliquées, le labyrinthe est généré aléatoirement.

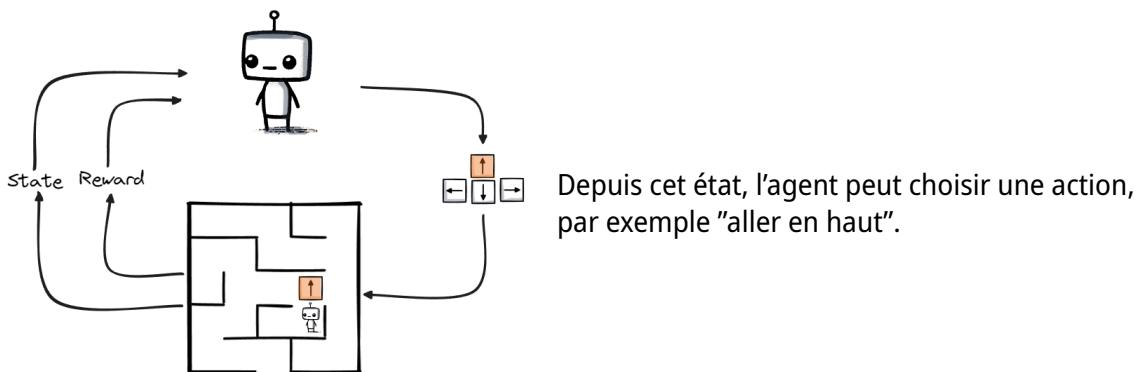
Bien sûr, nous pourrions pour cela construire un algorithme de recherche du chemin le plus court basé sur la théorie des graphes, mais cela reviendrait à coder explicitement le comportement de la machine. Et c'est là tout l'inverse du machine learning. À la place, nous voulons programmer la machine pour qu'elle développe elle-même la stratégie lui permettant de quitter le labyrinthe.

Nous allons donc utiliser une approche d'apprentissage par renforcement. Notre agent sera le petit robot, et le labyrinthe l'environnement dans lequel il évolue.

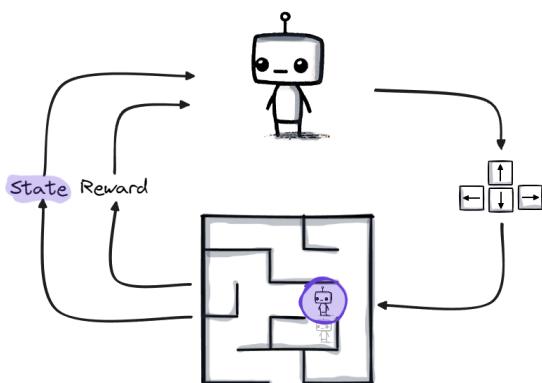


L'état (*State*) correspond à la position de l'agent dans le labyrinthe, et les actions à ses déplacements possibles (haut, bas, droite, gauche, droite)

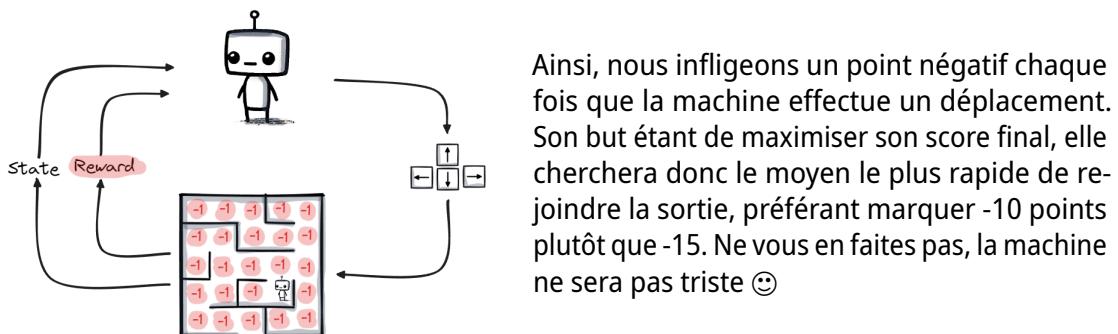




En effectuant cette action, l'agent change son état, ce qui s'accompagne d'une récompense. Ici, nous attribuons à la machine un score de -1 point.



Quoi?! Mais l'agent s'est pourtant rapproché de la sortie, pourquoi le sanctionner avec un score négatif? C'est vrai, cependant, rappelez-vous de notre objectif : "trouver la sortie le plus vite possible".



En résumé, notre objectif est d'apprendre à la machine, *quelle action effectuer lorsqu'elle se situe dans un état donné*. Autrement dit, nous cherchons à développer une fonction $f(s) = a$ qui prend en entrée l'état s et produit une action a . Cette fonction est couramment nommée **politique d'action**.

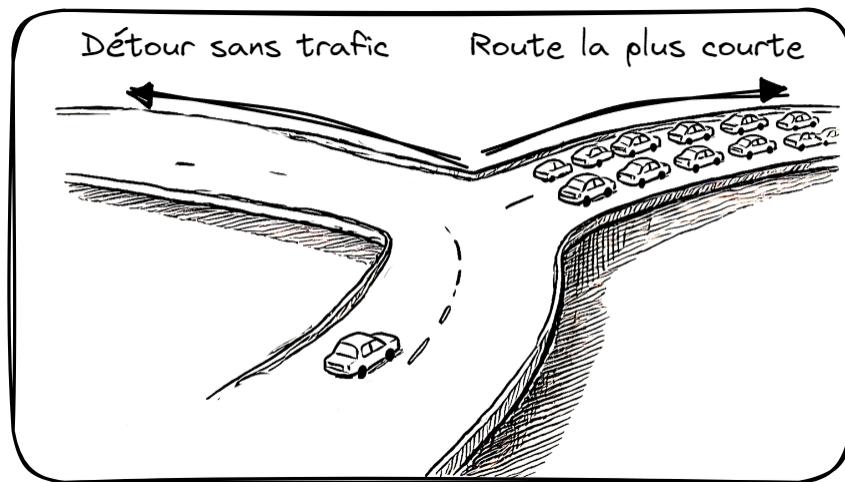
En apprentissage par renforcement, il existe plusieurs méthodes permettant d'apprendre à développer cette politique d'action. Celle que nous allons voir ici est l'approche appelée **Q-Learning**.

Le fonctionnement du Q-Learning

Le Q-Learning consiste à apprendre quelle est la **valeur** d'une action A dans un état S donné.

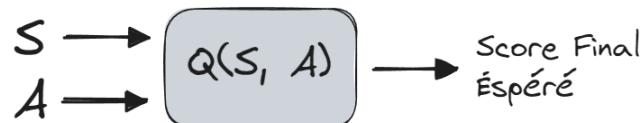
Par exemple, je mets habituellement 30 minutes en voiture pour aller à mon travail. Aujourd'hui, cette route est encombrée par le trafic. J'aperçois cependant qu'une autre route, un peu plus longue, est également possible. Ainsi, deux choix s'offrent à moi : l'action de tourner à gauche ou l'action de tourner à droite.

Pour choisir laquelle emprunter, je vais évaluer la valeur de chacune. À mon avis, quel sera le temps de trajet total si je tourne à gauche ? Et si je tourne à droite ?

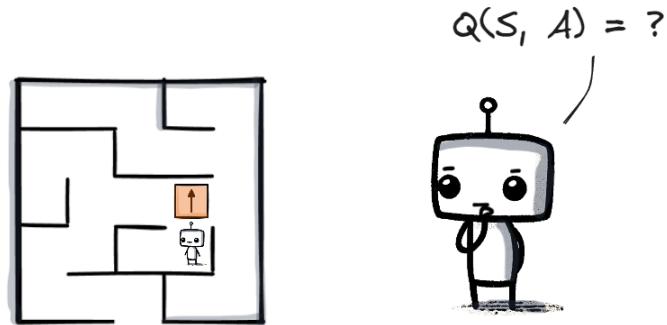


Voilà le but du Q-Learning : apprendre à prédire la valeur de mes actions, en tenant compte de la situation dans laquelle je me trouve.

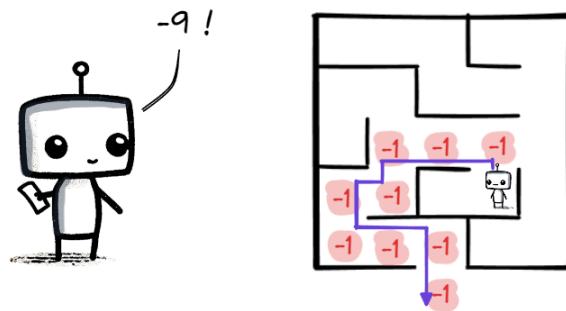
Plus formellement, il s'agit donc d'apprendre une fonction Q qui prend en entrée un état S et une action A, et qui prédit le **score final** que l'on peut espérer obtenir si tout se passe pour le mieux par la suite.



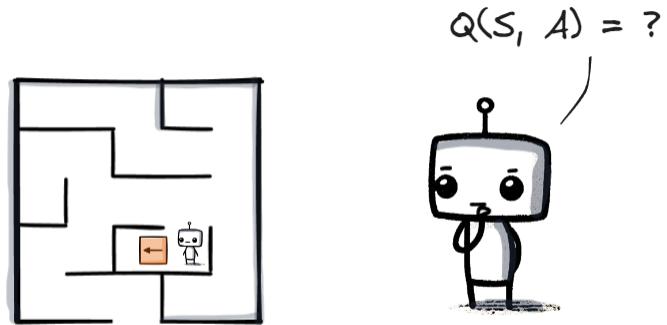
Prenons un exemple : dans la position actuelle, quel est le meilleur score que la machine puisse espérer obtenir si elle choisit l'action "allez en haut"?



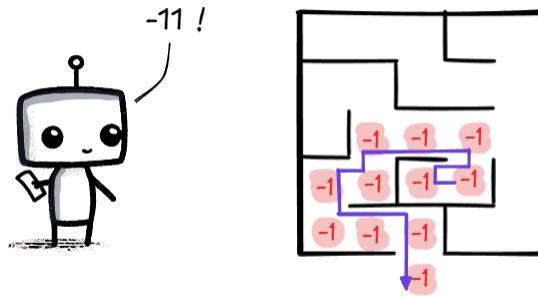
En tenant compte du fait que la machine perd un point pour chaque déplacement, le meilleur score qu'elle puisse espérer obtenir est -9.



Et dans le cas où la machine choisit d'aller à gauche?



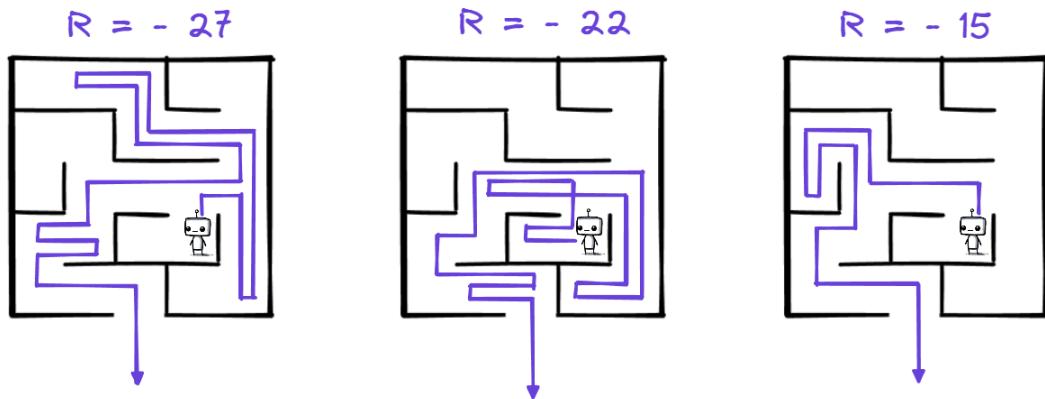
... Cette fois-ci, la réponse est -11, car la machine devra revenir sur son chemin. Ainsi, il est préférable de choisir l'action "aller en haut" car elle donne un score final plus élevé (-9) que l'action "aller à gauche" (-11)



Il reste maintenant une question : Comment apprendre cette fonction Q ?

L'apprentissage de la fonction Q

Pour apprendre à prédire la valeur de ses actions, la machine doit explorer son environnement en s'y déplaçant au hasard, afin de générer des données (S, A, R) (état, action, récompense).



Ces données sont ensuite utilisées pour mettre à jour un tableau $Q(S, A)$ qui informe la machine des récompenses qu'elle est censée obtenir à l'avenir en choisissant telle ou telle action dans l'état présent.

Au début de son apprentissage, ce tableau est rempli de valeurs aléatoires, comme illustré ci-dessous.

$Q(S, A)$

$S \setminus A$	\uparrow	\rightarrow	\downarrow	\leftarrow
	0.34	-0.18	1.69	-0.45
	0.81	-0.74	0.98	0.49
	-0.63	-0.90	0.29	0.73
⋮	⋮	⋮	⋮	⋮

Une formule, connue sous le nom de **l'équation de Bellman**, permet d'exprimer le score total que l'on peut espérer obtenir à l'avenir. Ce score est égal à la récompense obtenue immédiatement dans l'état S , à laquelle on ajoute le même score pour l'état suivant S' , si l'on choisit la meilleure action possible dans ce nouvel état.

$$Q(S, A) = E [R(S') + \gamma \max Q(S', A')]$$

Récompense immédiate
obtenue en allant à l'état S'
Récompenses
totales espérées dans
le nouvel état S' , en prenant
la meilleure action A'

Récompenses
totales espérées
Espérance
Mathématique
dévaluation

Pour bien comprendre, reprenons l'analogie avec la voiture.

L'équation de Bellman vous permet d'estimer que la valeur de tourner à gauche vous donne la récompense immédiate d'éviter le trafic, en plus de la valeur que vous allez obtenir dans votre prochain état. Si vous tombez sur des bouchons 1 km plus loin, le fait de tourner à gauche n'était peut-être pas la meilleure option.

À chaque fois que la machine réalise une action, elle peut mettre à jour sa table Q, car elle reçoit une récompense associée à son changement d'état. D'après la formule de Bellman, cette récompense $R(S')$ peut être utilisée pour recalculer la valeur de $Q(S,A)$ en se basant sur sa valeur actuelle et cette nouvelle donnée :

$$Q(S, A) = (1 - \eta) Q(S, A) + \eta (R(S') + \gamma \max Q(S', A'))$$

learning rate

Nouvelle valeur Q	Valeur Q actuelle	Equation de Bellman
---------------------	---------------------	---------------------

Exploration et exploitation

Cependant, dès que la table-Q commence à donner de bons résultats, la machine peut tomber dans le piège de suivre une politique d'action limitée, ignorant au passage les autres possibilités encore inexplorées.

De la même manière qu'après avoir trouvé un bon restaurant dans votre ville, vous choisissez systématiquement d'aller dîner dans ce restaurant car vous pensez qu'il n'y en a pas de meilleurs.

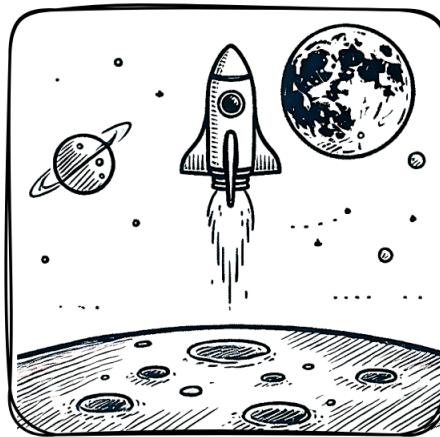
Pour éviter ce phénomène, on définit dans l’algorithme du Q-learning une probabilité **epsilon** qui pousse la machine à continuer **d’explorer** l’environnement des possibles, plutôt que de simplement **exploiter** sa politique d’action déjà apprise.

C'est là le dernier élément qui compose l'algorithme du Q-Learning.

Pour finir ce chapitre, je vous propose donc de passer à l'action avec une implémentation des plus amusantes 😊

Implémentation Python

Dans cette partie, nous allons développer un algorithme de Q-Learning dont le but sera d'apprendre à poser un vaisseau spatial sur la lune sans qu'il ne s'écrase.



Pour cela, nous aurons besoin d'un environnement (la lune) ainsi que d'un agent (le vaisseau spatial). Il existe de nombreuses librairies permettant de générer de tels environnements, l'une des plus populaires étant la librairie **gymnasium** de OpenAI.

Avant de commencer, il vous faudra donc créer un nouvel **environnement virtuel** pour y installer le package *gymnasium* dans sa version 0.29.1 ainsi que son extension *gymnasium[box2d]*.

Si vous rencontrez des difficultés lors de l'installation, les lignes de commandes suivantes devraient permettre de tout installer pour la plupart des utilisateurs :

1. pip install wheel setuptools pip –upgrade
2. pip install swig
3. pip install gymnasium==0.29.1
4. pip install gymnasium[box2d]

Une fois que vos packages sont bien installés, le code ci-dessous devrait fonctionner et vous montrer un joli petit vaisseau spatial.

```

1 import gymnasium as gym
2 env = gym.make("LunarLander-v2", render_mode="human")
3 observation, info = env.reset(seed=42)
4
5 for _ in range(1000):
6     action = env.action_space.sample() # this is where you would insert your policy
7     observation, reward, terminated, truncated, info = env.step(action)
8
9     if terminated or truncated:
10         observation, info = env.reset()
11
12 env.close()

```

Dans ce projet, le vaisseau spatial dispose de **quatre actions** possibles :

- Ne rien faire
- Activer le réacteur du bas, afin de ralentir sa descente
- Activer le réacteur de gauche, afin s'incliner à droite
- Activer le réacteur de droite, afin de s'incliner à gauche

Son **état** lui est décrit pas sa position dans l'espace, ainsi que sa vitesse, son angle, et son contact avec le sol.

Pour finir, le système de **récompenses** est défini par plusieurs règles, parmi lesquelles :

- Le vaisseau gagne 100 points s'il réussit à se poser sans dégâts
- Le vaisseau perd 100 points s'il se crash
- Le vaisseau perd des points chaque fois qu'il active un moteur

Nous allons donc commencer par initialiser notre environnement lunaire, ainsi que les différents hyper-paramètres de notre modèle : le *learning rate*, le facteur de dévalorisation, la valeur epsilon, et le nombre d'épisodes que la machine va jouer durant son apprentissage.

```

1 env = gym.make("LunarLander-v2", render_mode="human")
2
3 # hyper-paramètres
4 alpha = 0.1
5 gamma = 0.99
6 epsilon = 1.0
7 epsilon_decay = 0.995
8 epsilon_min = 0.01
9 episodes = 1000
10 max_steps = 1000

```

Tout comme notre petit robot qui se déplaçait dans le labyrinthe d'une case à l'autre, nous allons considérer que le vaisseau spatial se déplace sur une grille de coordonnées. C'est pourquoi nous allons découper l'espace des états du vaisseau spatial en plusieurs parties, afin de regrouper tous ces états dans la table-Q (initialement remplie de 0).

```

1 state_bins = [np.linspace(-1.0, 1.0, 10) for _ in range(env.observation_space.shape
2     ↪ [0])]
2 n_bins = tuple(len(bins) + 1 for bins in state_bins)
3 q_table = np.zeros(n_bins + (env.action_space.n,))
4
5 def discretize_state(state):
6     return tuple(np.digitize(state[i], state_bins[i]) for i in range(len(state)))

```

Ensuite, nous allons créer une boucle pour entraîner notre vaisseau spatial à se poser sans dégât. Cette boucle comprendra 1000 épisodes, et chaque épisode comprend au maximum 1000 pas.

À chaque pas, le vaisseau spatial "lancer un dé", et en fonction du résultat, il choisira soit **d'explorer** son environnement en choisissant une action au hasard, soit **d'exploiter** sa politique d'action en choisissant l'action qui, selon lui, a la plus grande valeur (cette valeur est définie par la table-Q)

Une fois son action effectuée, le vaisseau recevra une récompense, qu'il utilisera pour mettre à jour la table-Q en appliquant la formule de mise à jour et l'équation de Bellman.

```

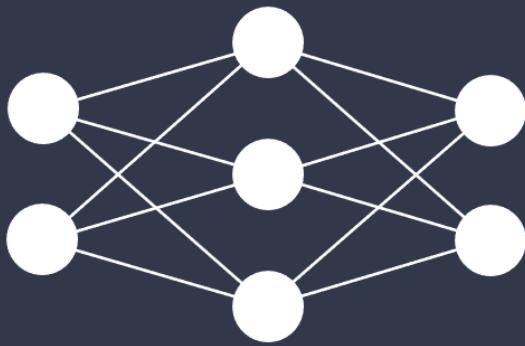
1  for episode in range(episodes):
2      state, _ = env.reset(seed=42)
3      state = discretize_state(state)
4      total_reward = 0
5
6      for step in range(max_steps):
7          # Epsilon-greedy policy
8          if random.uniform(0, 1) < epsilon:
9              action = env.action_space.sample() # Exploration
10         else:
11             action = np.argmax(q_table[state]) # Exploitation
12
13         next_state, reward, terminated, truncated, _ = env.step(action)
14         next_state = discretize_state(next_state)
15
16
17         old_value = q_table[state][action] # valeur Q actuelle
18         next_max = np.max(q_table[next_state]) # valeur Q max de l'état S'
19
20         # mise à jour de la table q avec la formule de Bellman
21         new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
22         q_table[state][action] = new_value
23
24         state = next_state
25         total_reward += reward
26
27         if terminated or truncated:
28             break
29
30     # Reduction de Epsilon
31     if epsilon > epsilon_min:
32         epsilon *= epsilon_decay
33
34     print(f"Episode {episode + 1}, Total Reward: {total_reward}")
35
36 env.close()

```



Bravo Commandant! Mission accomplie! 😊

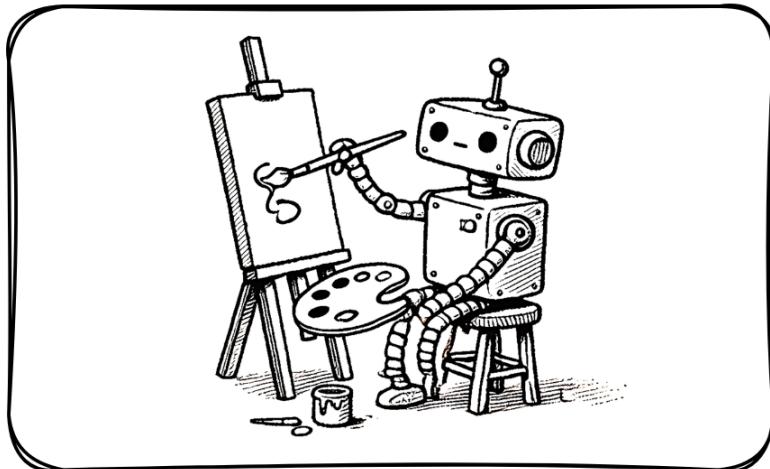
Generative AI



Deep learning et generative AI

Pour le dernier chapitre de ce livre, nous allons finir en beauté en abordant un domaine qui est aujourd'hui au summum du machine learning : l' **Generative AI**. Comme son nom l'indique, il s'agit du domaine dans lequel les modèles apprennent à générer du texte, des images, des vidéos, de la musique, etc.

Depuis la sortie de ChatGPT en novembre 2022, le monde entier a les yeux tournés vers cette discipline, et je suis heureux de vous en enseigner les bases ici!



Première question : comment fonctionne un modèle de GenAI ? S'agit-il d'apprentissage supervisé ? Non supervisé ? D'apprentissage par renforcement ? Bien souvent, les trois méthodes sont impliquées dans la création de ces modèles.

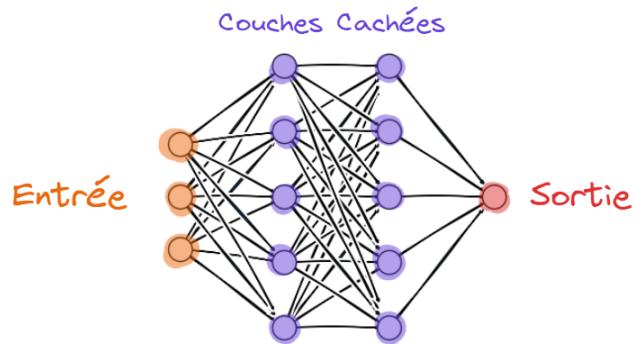
Cependant, il existe un socle commun à toutes les techniques modernes de GenAI : celle du deep learning. C'est donc par là que nous allons commencer.

Qu'est-ce-que le deep learning ?

Le deep learning est une discipline très semblable à celle du machine learning. Il s'agit en effet du même principe, à savoir entraîner des modèles à partir de données.

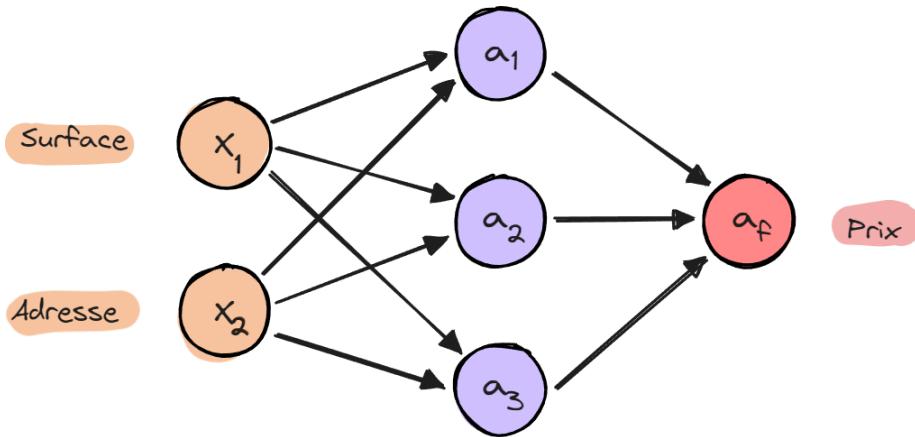
La principale différence réside dans le type de modèle que l'on entraîne. En effet, le deep learning se focalise uniquement sur la création de modèles connus sous le nom de **réseaux de neurones artificiels**

Un réseau de neurones est un modèle composé d'un grand nombre de petites fonctions non linéaires, connectées les unes aux autres à travers une organisation en couches.

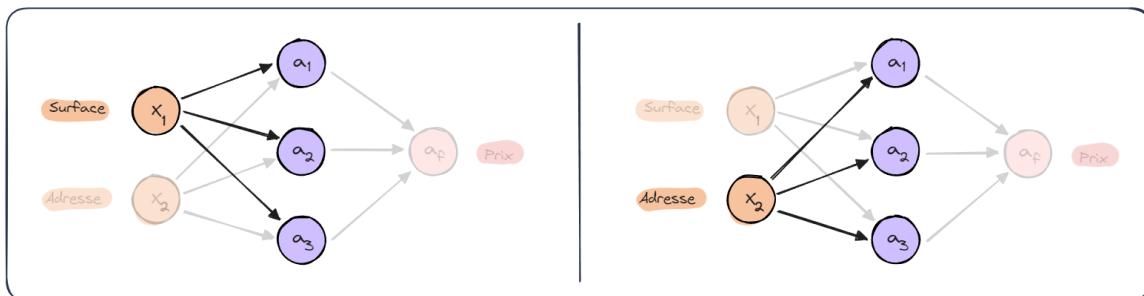


Ces modèles peuvent aussi bien être utilisés pour résoudre des problèmes de régression, de classification, de clustering, ou même d'apprentissage par renforcement.

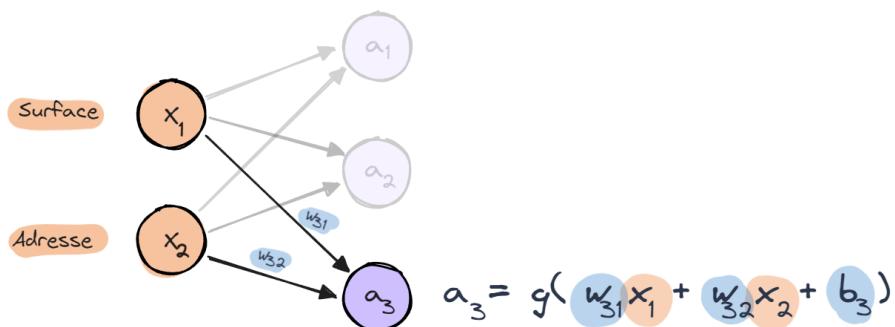
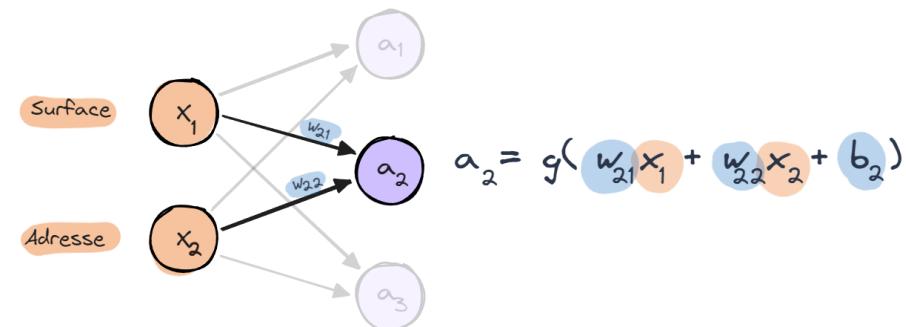
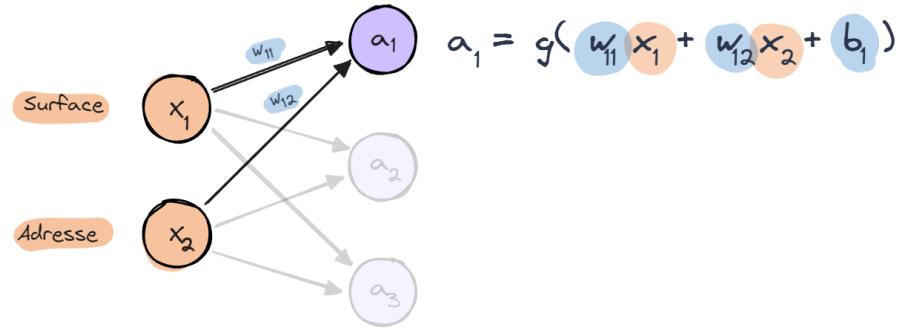
Par exemple, voici un réseau de neurones traitant des données sur immobilières pour prédire le prix d'une maison, comme nous l'avions vu dans le chapitre 2.



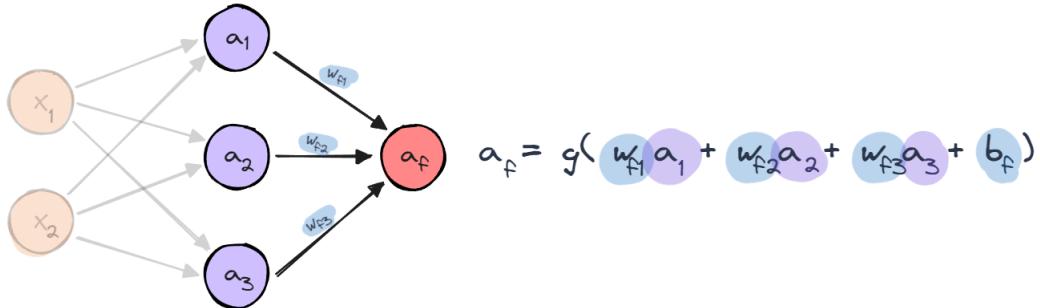
Dans un premier temps, chaque variable d'entrée est envoyée vers les neurones de la première couche.



Les neurones combinent ces entrées en les multipliant par un poids w , de manière semblable à ce que nous avons vu dans le chapitre sur la régression linéaire. Ensuite, cette combinaison linéaire est transmise à une fonction d'activation g . Cette fonction peut varier d'une application à l'autre, mais en général, elle permet au neurone de s'activer et de produire une sortie lorsque ses entrées dépassent un certain seuil.



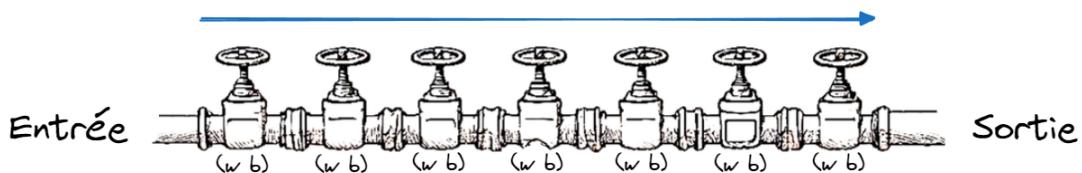
Pour finir, les résultats de ces neurones sont envoyés à la couche suivante, en suivant le même principe de combinaison linéaire et de fonction d'activation.



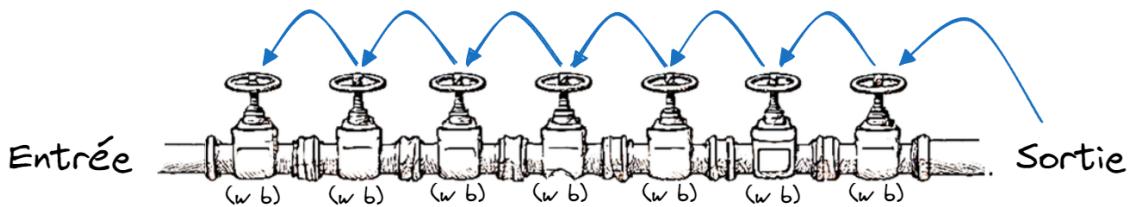
Ici, notre réseau ne comporte que deux couches, mais on peut ajouter autant de couches et de cellules que l'on souhaite au sein de notre modèle. Plus on rajoutera de couches, plus le modèle deviendra puissant (mais également plus coûteux à entraîner)

En parlant d'entraînement, un réseau de neurones utilise généralement la méthode de la descente de gradient (vue dans le chapitre 2) pour apprendre à prédire correctement la sortie y attendue. Mais contrairement à une simple régression linéaire, nous avons ici une chaîne de fonctions où le comportement des premières influence celui des suivantes, rendant le réglage des paramètres (w, b) plus complexe.

Pour utiliser une analogie, cela revient à vouloir contrôler le débit d'eau y en sortie d'un robinet, en réglant une série de valves (w, b) situées le long de la canalisation. Chaque fois que l'on tourne une valve, cela affecte l'eau qui passe à travers toutes les valves suivantes. Alors comment procéder ?



La réponse est de partir de la sortie y et de remonter jusqu'à l'entrée X , en comprenant comment chaque valve est impactée par celle qui la précède. Ce processus se calcule à l'aide d'un gradient (le même gradient que celui utilisé dans le chapitre 2) ce qui nous permet de créer une chaîne de gradients. Ces gradients sont ensuite utilisés pour ajuster les paramètres de chaque couche grâce à l'algorithme de la descente de gradient. Tout ce processus porte un nom : la **Back-Propagation**.



Si vous souhaitez en savoir plus à ce sujet, je vous invite à consulter ma série de vidéos YouTube sur le deep learning, déjà visionnée par plus de 1 million de personnes depuis sa création. Elle vous permettra de plonger en profondeur dans cet univers passionnant!

Les différentes architectures de réseaux de neurones

Les réseaux de neurones artificiels peuvent adopter de nombreuses architectures différentes, chacune étant associée à une famille d'applications. Parmi elles, on retrouve :

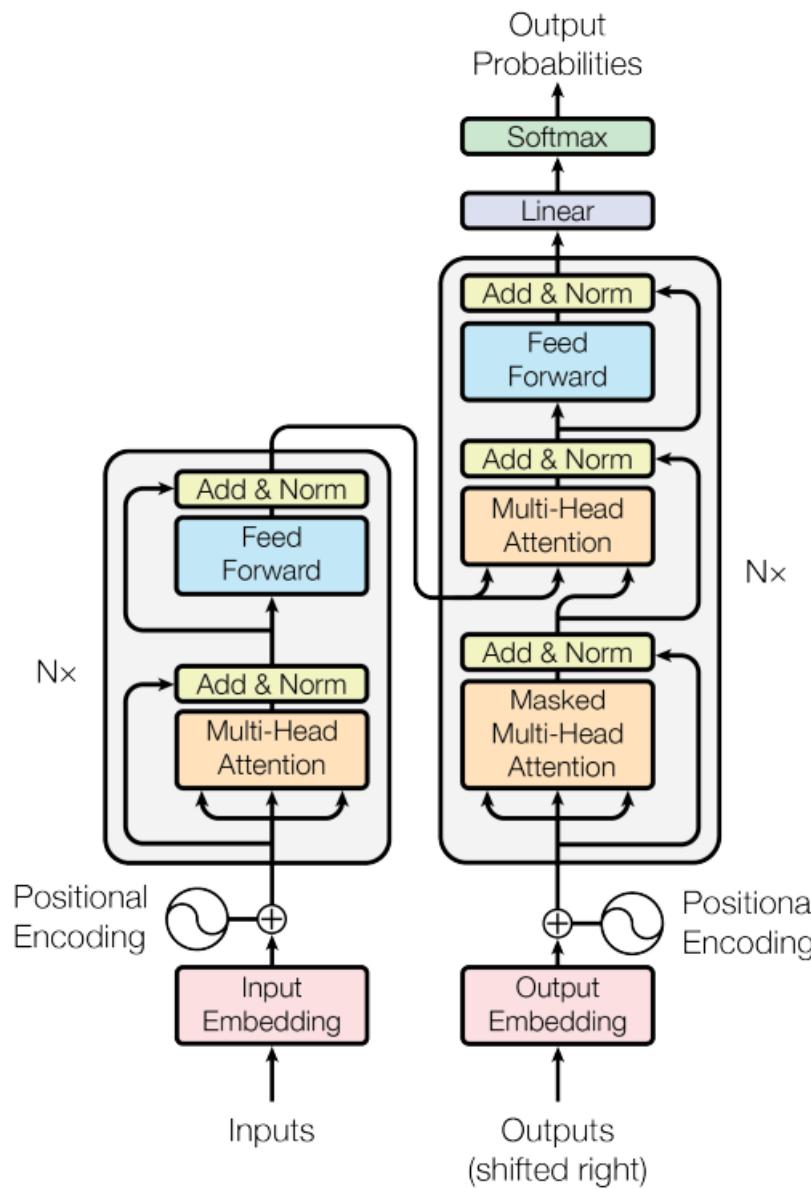
- Les **Réseaux de Neurones à Convolution** (CNN) : utilisés pour les applications liées à l'image, comme le traitement d'image, la détection d'objets, la reconnaissance faciale, etc.
- Les **Réseaux de Neurones Récurrents** (RNN) : utilisés pour les applications liées aux séquences, telles que les séries temporelles, l'analyse de texte, la musique, etc.
- les **Auto-Encodeurs**, utilisés pour le clustering, la réduction de dimension, mais aussi les deep fakes.
- ...

Dans ce chapitre, nous allons nous intéresser à l'une de ces architectures, qui a révolutionné le monde de l'IA depuis sa sortie en 2017 : le **transformer**.

Le transformer

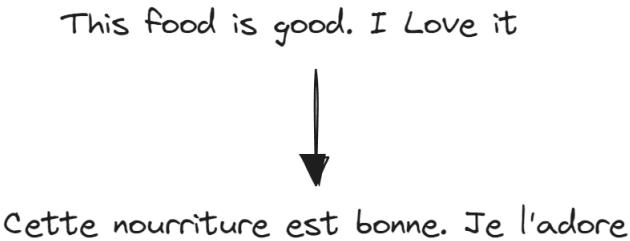
Inventé en 2017 par Google, le *transformer* est l'architecture à l'origine de ChatGPT et de tous les autres *Large Language Models* qui prolifèrent aujourd'hui dans le monde de l'IA.

Voici son architecture, dont l'image est tirée de l'article d'origine : ([disponible ici](#))



Ne vous en faites pas, je vais tout vous expliquer.

Imaginez vouloir développer un modèle de traduction Anglais → Français.



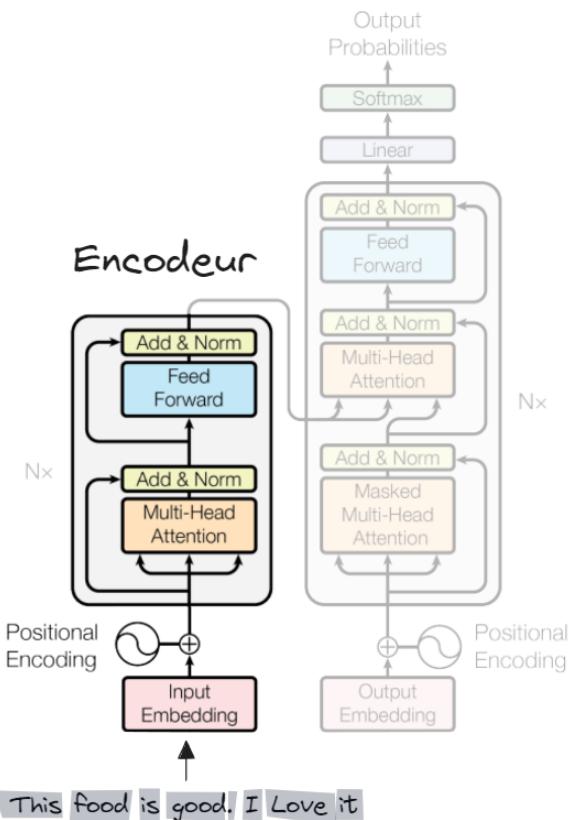
La première étape est de fournir la phrase d'entrée dans la partie **Encodeur** de notre modèle.

Pour cela, on commence par découper notre phrase en petit fragments, appelés **tokens**.

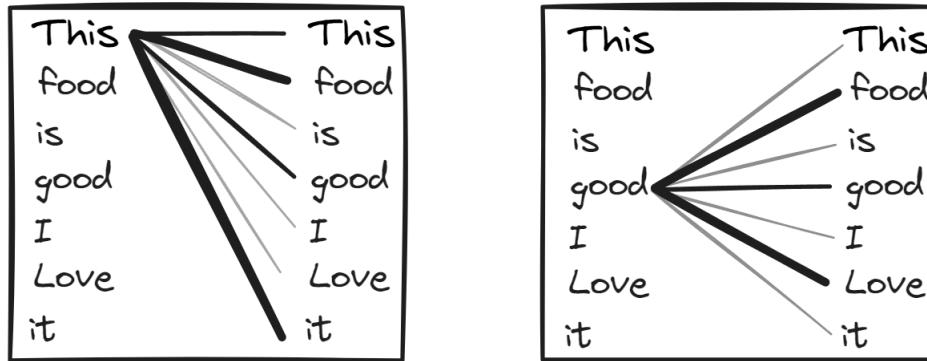
This food is good. I Love it

Ces fragments sont ensuite convertis en vecteurs, c'est la partie **Embedding**. Ces vecteurs offrent une représentation numérique des mots de notre langue. Par exemple, les mots "food", "restaurant", "burger", "meal" sont des mots dont les vecteurs sont très proches les uns des autres. Cela permet à la machine de comprendre que tous ces mots appartiennent à une même thématique.

Ensuite, ces vecteurs traversent un mécanisme appelé **Attention**, qui consiste à apprendre quels mots sont reliés les uns aux autres dans une phrase.

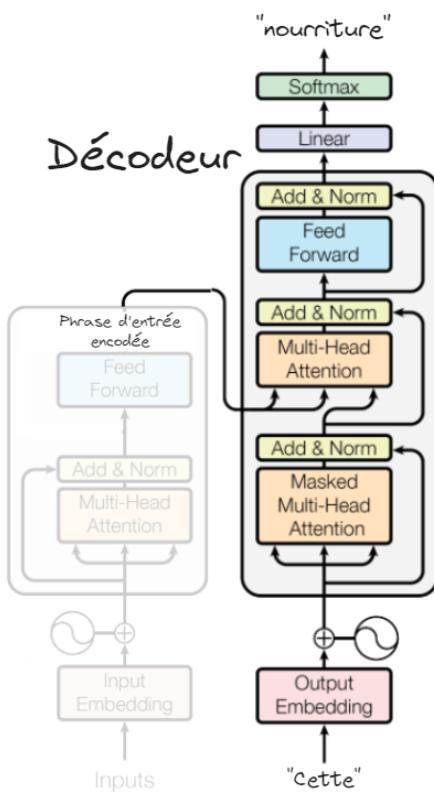


Par exemple, le modèle apprend à travers ses paramètres (w, b) que le mot "This" est relié aux mots "Food" et "it" dans le contexte de cette phrase. De la même manière, "good" est en connexion avec le mot "food" et "love".



Pour finir, le résultat de ces couches d'attention passe dans un réseau de neurones (**Feed Forward**), c'est-à-dire un réseau semblable à celui que nous avons vu au début de ce chapitre. Cela permet de transformer la phrase d'entrée en une matrice de valeurs représentant le sens des mots, leur position, et leurs connexions les uns avec les autres. Nous avons ainsi **encodé** notre phrase.

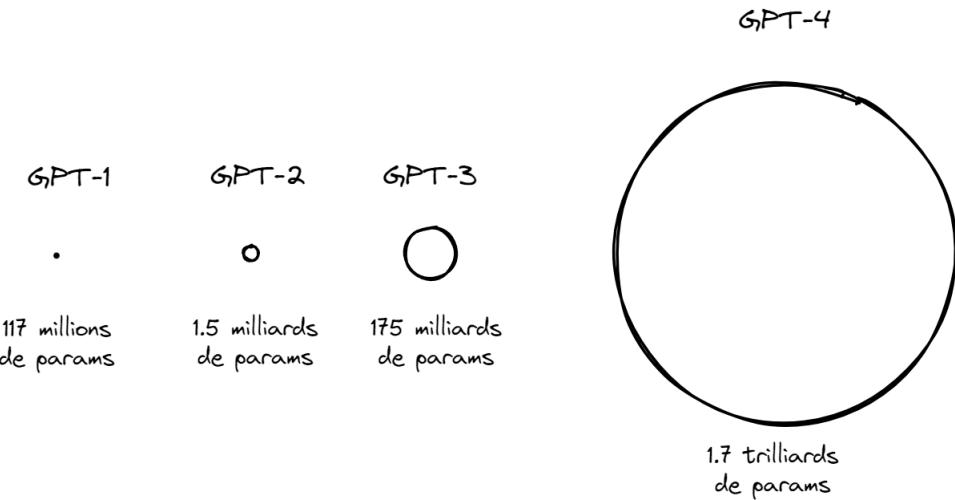
L'étape suivante est donc tout naturellement celle du **décodage**.



La matrice créée par l'encodeur est ensuite envoyée vers le **décodeur** qui va jouer le rôle de générateur de texte.

Chaque fois qu'il génère un nouveau mot, celui-ci est ajouté à l'entrée de son décodeur, et les mêmes mécanismes que ceux vus précédemment sont utilisés pour continuer de produire des mots en tenant compte à la fois du contexte fourni par l'entrée et de ce qui a été généré jusqu'à présent.

Tout ce beau monde fonctionne grâce à des millions, voir des milliards de paramètres (w, b). Pour développer ces modèles, des années d'entraînement ont été nécessaires, utilisant des bases de données textuelles parfois équivalentes à tout le contenu disponible sur Internet. Voici, à titre indicatif, l'évolution du nombre de paramètres des modèles GPT d'OpenAI depuis la sortie de la première version en 2018.



Heureusement, il n'est pas nécessaire de recoder un transformeur à partir de zéro. De nombreuses librairies proposent aujourd'hui des versions open source de plusieurs *Large Language Models*. Pour conclure ce livre en beauté, je vous propose de créer en quelques lignes de code seulement, un modèle de GenAI basé sur GPT-2.

Implémentation Python

Pour utiliser notre modèle de GPT-2, nous allons simplement le charger depuis le package *transformer*, disponible sur HuggingFace. Pour installer ce package, il vous suffit de l'ajouter à votre environnement virtuel Python, ou bien d'utiliser Google Colab si vous rencontrez des difficultés.

Le code qui suit est très simple : il charge un modèle GPT-2, puis l'utilise pour générer du texte, en suivant le prompt de votre choix.

```

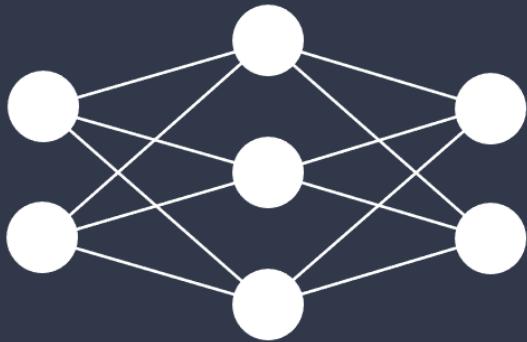
1 from transformers import pipeline, set_seed
2 generator = pipeline('text-generation', model='gpt2')
3 set_seed(42)
4 generator("I loved reading that book.", max_length=50, num_return_sequences=1)

```

La sortie générée par le modèle :

"I loved reading that book, but what I realized in that short span of time was that it could be a very powerful, powerful way to develop what I like to call'self-awareness'."

Conclusion



Conclusion

Félicitations !

Vous êtes arrivé au bout de ce voyage pour apprendre les bases du machine learning en une semaine. J'espère que ce livre vous a plu, tant par sa pédagogie et ses illustrations, que par les exemples de codes que j'ai composés pour vous.

Depuis la sortie de ChatGPT en novembre 2022, le monde connaît un bouleversement sans précédent dans le domaine de l'IA. Les entreprises de toutes tailles et de tous secteurs s'appuient de plus en plus sur l'intelligence artificielle pour prendre des décisions éclairées, automatiser des processus, et découvrir des opportunités cachées. En vous formant au machine learning, vous vous placez à l'avant-garde de cette révolution, ouvrant des portes vers des carrières passionnantes, des projets innovants, et un impact réel sur le monde.

Imaginez-vous, doté de ces compétences, capable de créer des solutions qui transforment non seulement des entreprises, mais aussi des vies. En vous formant à l'IA, vous ne ferez plus parti des suiveurs, mais parti des architectes du monde de demain.

Pour aller plus loin...

Si ce voyage vous a inspiré et que vous souhaitez aller plus loin, je vous invite à découvrir mes formations avancées. Conçues pour vous aider à approfondir vos connaissances et à appliquer concrètement ce que vous avez appris, ces formations sont la suite logique de votre parcours. Elles vous permettront d'acquérir les concepts mathématiques essentiels à connaître en IA, avoir une parfaite maîtrise des outils de programmation, et de travailler sur des projets réels, avec un accompagnement personnalisé.

Tout cela vous permettra de passer du stade d'apprenant à celui d'expert.

Pour en savoir plus et réserver votre place, rendez-vous sur machinelearnia.com/formations.

L'apprentissage ne s'arrête jamais, et le chemin devant vous est encore rempli de découvertes fascinantes. Je suis honoré de vous avoir accompagné jusqu'ici, et j'espère continuer à le faire dans cette prochaine étape de votre évolution.

Merci à vous 😊

Guillaume