Introduction à l'Intelligence Artificielle

I. Présentation de l'Intelligence Artificielle.

I.1. Origine et but de l'IA.

Le début de l'Intelligence Artificielle a lieu dans les années 50 avec les travaux d'Alan Turing.
Durant les années 80, l'IA est utilisée dans les systèmes experts (l'exemple connu de l'ordinateur DeepBlue qui bat les meilleurs joueurs d'échecs).
Depuis 2011, l'IA connait une progression spectaculaire grâce à l'apparition de processeurs à bas coût avec des puissances de calcul très élevées et un accès rapide au Big Data.
Le but est de reproduire les comportements du vivant qu'il soit naturel ou créé. On distingue deux types d'IA :

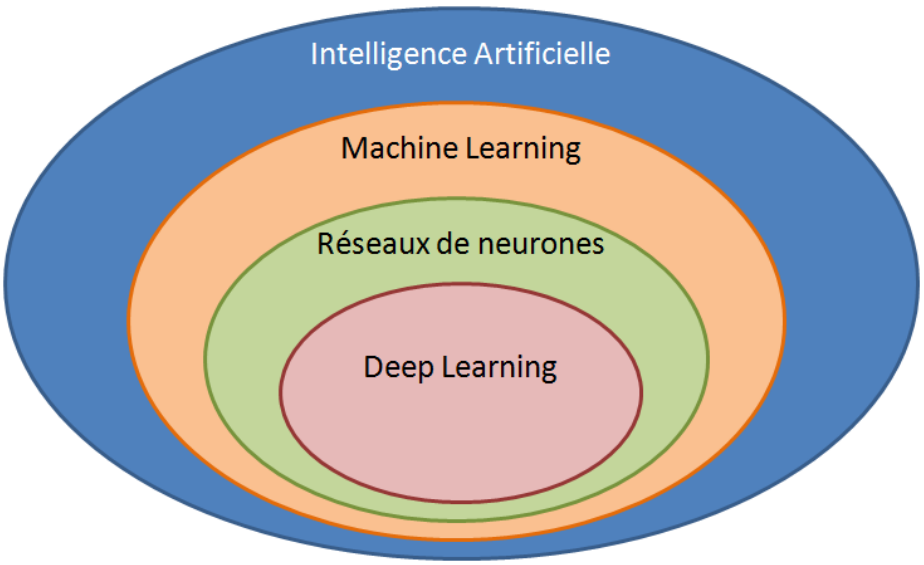
- L'IA "Forte" qui possède une conscience propre, mais n'existe pas ou pas encore.
- L'IA "Faible" avec des problèmes spécifiques sans réelle autonomie, ni conscience propre.

Quelques exemples d'application de l'IA :

- assistance vocale,
- jeux de stratégie,
- traduction automatique,
- reconnaissance automatique,
- véhicule autonome,
- détection de fraude,
- contrôle qualité,
- médecine prédictive,...

I.2. Apprentissage automatique

Les progrès récents en IA sont des progrès sur les Machines Learning (ML) ou apprentissage automatique. Cela consiste à réaliser une tâche spécifique en se basant sur un grand nombre d'exemples. Il s'agit de développer un modèle prédictif à partir d'un grand nombre de données.
Le Deep Learning (DL) ou apprentissage profond est une branche du Machine Learning qui utilise des réseaux de neurones.
Le schéma ci-contre permet de classer les différents niveaux de l'IA.



Les applications suivantes utilisent l'apprentissage profond :

- traitement automatique d'image (reconnaissance d'un élément particulier sur une photo, ...),
- traitement automatique du langage,
- génération automatique de texte, images,...

I.3. Différents modes d'apprentissage.

Il existe trois modes d'apprentissage : supervisé, non supervisé ou par renforcement.

- L'**apprentissage supervisé** avec la classification qui permet de labelliser des objets comme des images et la régression qui permet de réaliser des prévisions sur des valeurs numériques. On peut aussi parler d'entrées pour les objets et de sortie pour les labels. L'apprentissage est supervisé car il exploite des bases de données d'entraînement qui

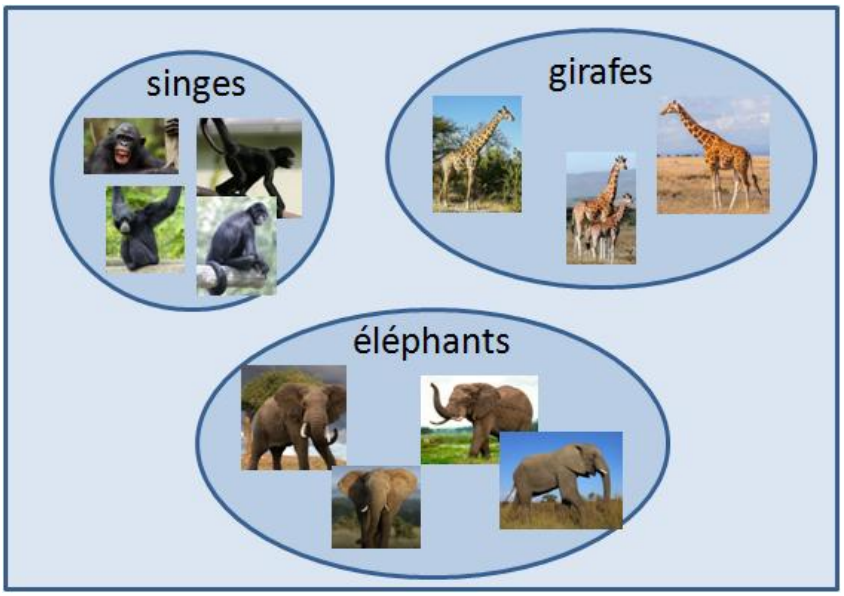
contiennent des labels ou des données contenant les réponses aux questions que l'on se pose. En gros, le système exploite des exemples et acquiert la capacité à les généraliser ensuite sur de nouvelles données de production.

- L'**apprentissage non supervisé** avec le clustering et la réduction de dimensions. Il exploite des bases de données non labellisées. Ce n'est pas un équivalent fonctionnel de l'apprentissage supervisé qui serait automatique. Ses fonctions sont différentes. Le clustering permet d'isoler des segments de données spatialement séparés entre eux, mais sans que le système donne un nom ou une explication de ces clusters. La réduction de dimensions (ou embedding) vise à réduire la dimension de l'espace des données, en choisissant les dimensions les plus pertinentes. Du fait de l'arrivée des big data, la dimension des données a explosé et les recherches sur les techniques d'embedding sont très actives.
- L'**apprentissage par renforcement** pour l'ajustement de modèles déjà entraînés en fonction des réactions de l'environnement. C'est une forme d'apprentissage supervisé incrémental qui utilise des données arrivant au fil de l'eau pour modifier le comportement du système. C'est utilisé par exemple en robotique, dans les jeux ou dans les chatbots capables de s'améliorer en fonction des réactions des utilisateurs. Et le plus souvent, avec le sous ensemble du Machine Learning qu'est le Deep Learning. L'une des variantes de l'apprentissage par renforcement est l'apprentissage supervisé autonome notamment utilisé en robotique où l'IA entraîne son modèle en déclenchant d'elle-même un jeu d'actions pour vérifier ensuite leur résultat et ajuster son comportement.

Les différences entre ces trois apprentissages sont importantes, mais seuls les apprentissages supervisé et non supervisé sont au programme de CPGE.

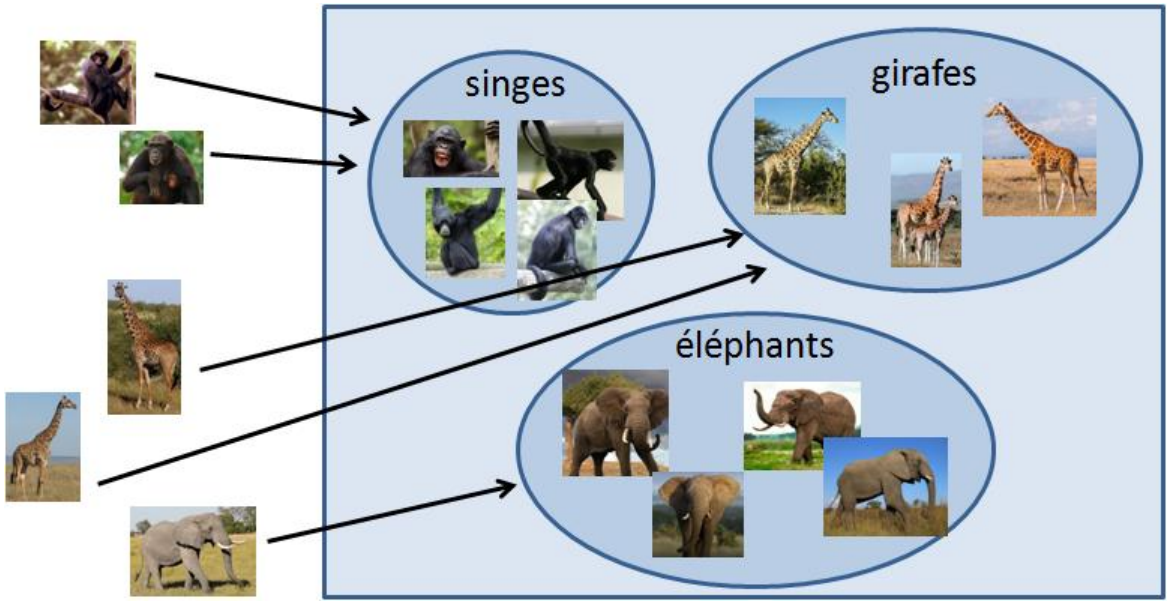
Exemple d'apprentissage supervisé

On fournit à l'IA un ensemble d'images (correspondant aux objets ou entrées) associées au nom de l'animal concerné (correspondant aux labels ou sorties).



L'IA par apprentissage supervisé reconnait l'appartenance des nouvelles images en fonction de leurs caractéristiques.

Ici, elle reconnait que parmi les cinq nouvelles images, deux correspondent au label "singes", une au label "éléphants" et deux au label "girafes".



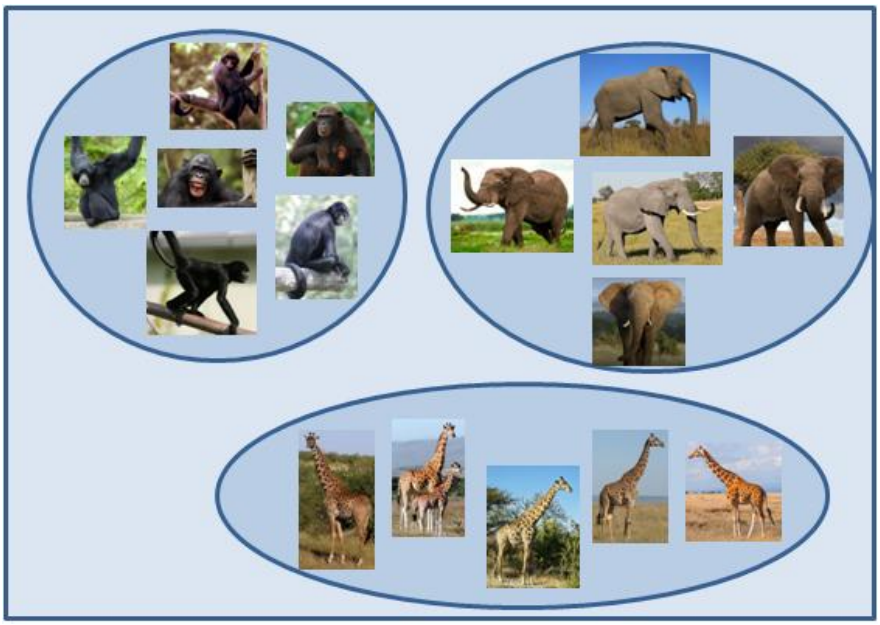
Exemple d'apprentissage non supervisé

On fournit à l'IA une ensemble d'éléments qu'elle va regrouper ressemblance :

Eléments fournis à l'IA : un ensemble d'images représentant des animaux

L'IA regroupe les animaux en fonction des caractéristiques communes (couleurs, formes,...).
Aucun label (ici les noms des animaux) n'est défini.

.



I.4. Principes généraux de l'apprentissage supervisé.

Il existe deux types de problème à traiter : les régressions et les classifications. La différence entre les deux porte essentiellement sur le type de sorties souhaitées.
Pour les algorithmes de régression, la sortie peut prendre une infinité de valeurs dans l'ensemble continu des réels.
Pour les algorithmes de classification, la sortie ne peut prendre qu'un nombre fini de valeurs que l'on nomme étiquettes. On parlera de classification binaire lorsque la sortie ne peut prendre que deux valeurs et de classification multi-classe lorsqu'elle peut prendre plus de deux valeurs (couleurs, reconnaissance de panneaux routiers,...).

I.5. Définitions et notations.

On considère que les données sont représentées sous forme d'entrées et de sorties :

- L'entrée est représentée par un vecteur X de valeurs d'attributs réels. Par exemple, une image est représentée par un vecteur contenant la valeur de chaque pixel.

- La sortie ou cible *y* aura des représentations différentes en fonction du problème traité. Pour les problèmes de régression, la sortie prendra une valeur réelle. Pour les problèmes de classification en *C* classes, la sortie prendra une valeur discrète avec un index entre 0 et *C-1*.

Description de l'algorithme supervisé :

On note l'ensemble d'entraînement *D* de *m* éléments : $D=\{(X_1,y_1),(X_2,y_2),....(X_t,y_t),....(X_m,y_m)\}$. La relation entre les entrées X_t et sorties y_t est inconnue. On notera n le nombre d'éléments du vecteur X et m le nombre d'éléments de D.

Le but de l'apprentissage supervisé va constituer à déterminer une fonction prédictive *f* appelée modèle qui sera une bonne approximation de cette relation tel que : $y_t \approx f(X_t)$.

Les données d'entraînement seront indicées par la lettre *t* (training).

Pour quantifier la qualité de cette approximation, on définira une fonction coût *J* qui sera caractérisée par une distance entre la sortie du jeu d'entrainement et celle issue du modèle : $J(y_t, f(X_t))$.

Exemple de la prédiction du prix d'un logement.

On dispose du prix de vente d'un certain nombre de logements pour lesquels on connait la surface, le nombre de chambres et la qualité :

Surface en m²	Nombre de chambres	Qualité	Prix en euros
124	3	3	313 000
339	5	5	2 384 000
179	3	4	342 000
186	3	5	420 000
180	4	5	550 000
82	2	2	490 000
125	2	4	335 000

Le but de l'IA à développer consiste à prédire un prix de vente pour un appartement dont on connait la surface, la qualité et le nombre de chambres.

On a ici 7 échantillons donc $m=7$ dont chacun contiennent 3 informations, donc $n=3$. La matrice X possède $m=7$ lignes et $n=3$ colonnes. Le vecteur Y possède donc $m=7$ lignes :

$$X=\begin{pmatrix}124 & 3 & 3\\ 339 & 5 & 5\\ 179 & 3 & 4\\ 186 & 3 & 5\\ 180 & 4 & 5\\ 82 & 2 & 2\\ 125 & 2 & 4\end{pmatrix} \qquad Y=\begin{pmatrix}313000\\ 2384000\\ 342000\\ 420000\\ 550000\\ 490000\\ 335000\end{pmatrix}$$

L'objectif de l'apprentissage automatique est dans ce cas, de déterminer une relation, pas nécessairement explicite, soit un modèle entre X et Y : $Y=f(X)$ avec f la relation recherchée.

Très souvent, les données d'entrées sont normalisées, c'est à dire mises à la même échelle, le plus souvent entre -1 et 1. En effet, les données peuvent avoir des valeurs très différences et les algorithmes risquent de privilégier certaines données devant d'autres.

Sur l'exemple précédent, la matrice X normalisée devient :

$$X=\begin{pmatrix}0.366 & 0.6 & 0.6\\ 1.0 & 1.0 & 1.0\\ 0.528 & 0.6 & 0.8\\ 0.549 & 0.6 & 1.0\\ 0.531 & 0.8 & 1.0\\ 0.242 & 0.4 & 0.4\\ 0.369 & 0.4 & 0.8\end{pmatrix}$$

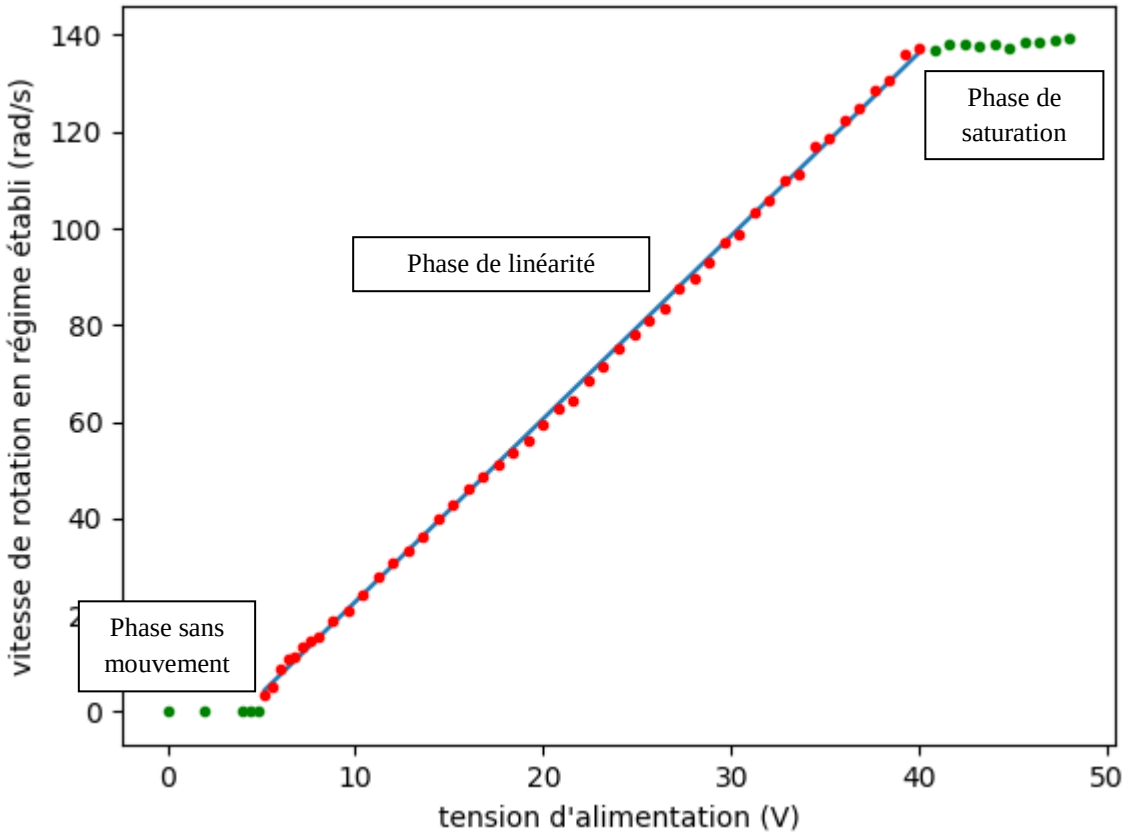
I.6. Problème de régression.

Pour un problème de régression, les sorties peuvent prendre un nombre infini de valeurs dans l'ensemble des réels : tailles, températures,... Il s'agit donc de donner une représentation mathématique sous la forme d'une droite (régression linéaire), d'un polynôme (régression polynomiale), par une fonction logarithmique,...

Exemple de régression linéaire : ci-dessous, on a représenté la vitesse de rotation en régime établi du moteur du système Control'X en fonction de la tension du moteur. On a effectué 64 mesures entre 0 et 48V. Les points rouges représentent les points expérimentaux sur la zone de linéarité, les 5 premiers points verts représentent les essais avec une tension insuffisante

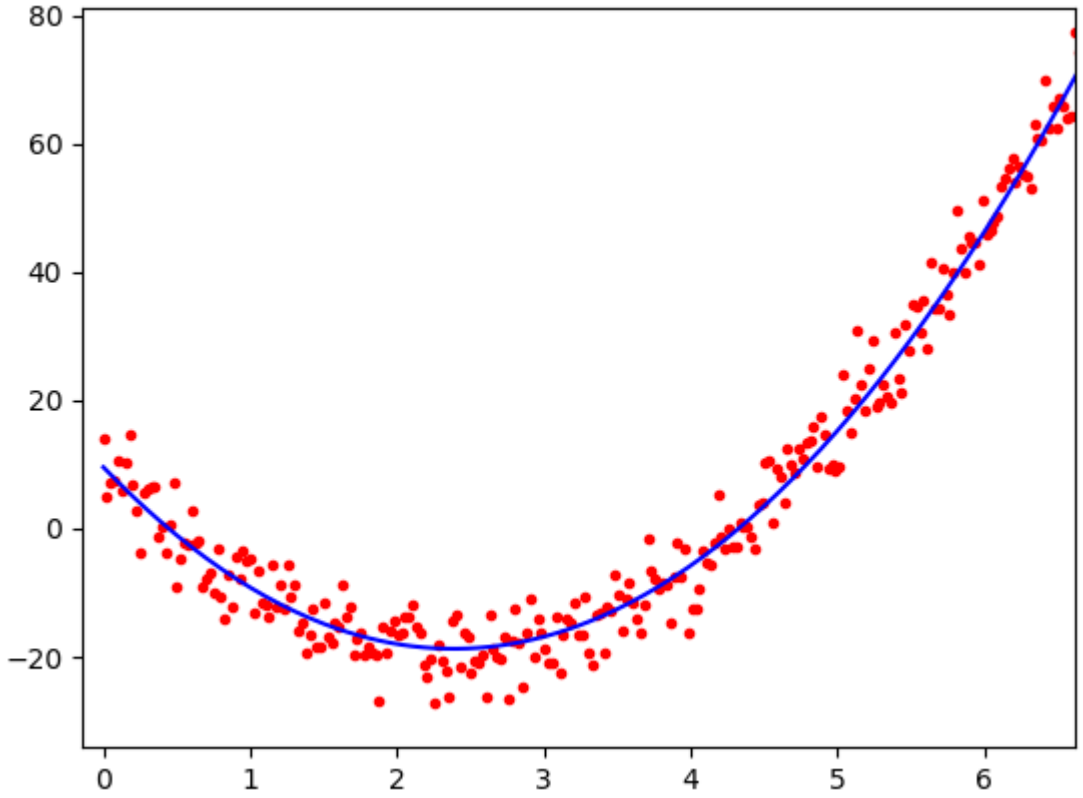
pour avoir un mouvement, 9 derniers points verts représentent les essais avec une tension supérieure à la tension de saturation (40V).

La régression linéaire tracée bleu a été effectuée sur la zone de linéarité (points rouges).



Exemple de régression linéaire représentant la vitesse de rotation en régime établi en fonction de la tension d'alimentation du moteur à courant continu pour le Control'X

Exemple de régression polynomiale de degré 2 :



Exemple de régression polynomiale de degré 2.

Les deux exemples précédents sont des exemples de régression mono-variable, les algorithmes de régression peuvent être multi-variables lorsque les données d'entrée possèdent plusieurs composantes.

On peut citer quelques exemples d'algorithmes de régression :

- Régression linéaire uni-variable ou multi-variable, au programme des CPGE
- Arbres de régression,
- Régression vectorielle de support,...

I.7. Problème de classification.

Un problème de classification consiste à prédire une classe parmi un ensemble fini. Par exemple, déterminer si une image contient un singe, une girafe,...

Il s'agit bien d'un problème d'apprentissage supervisé, car les classes sont bien connues à l'avance. Sur l'exemple présenté en début de poly, on a fournit à l'algorithme un certain nombre d'image de singes et l'algorithme définit si les images à tester appartiennent à cette classe. Sur cet exemple, le problème possède 3 classes : "singes", "éléphants" et "girafes".

On peut citer quelques exemples d'algorithmes de classification :

- Régression logistique,
- Arbres de classification (dont Forêts aléatoires),
- K plus proches voisins (KNN), au programme des CPGE
- Machine à vecteurs de support,
- Classification naïf de Bayes,
- Réseaux de neurones,.... au programme des CPGE

II. Formalisme des algorithmes.

II.1. Etapes des algorithmes d'apprentissages automatiques.

Tous les algorithmes d'apprentissages automatiques fonctionnent en plusieurs temps :

- La première phase consiste à récolter les données et à les mettre sous forme de la matrice X . Pour un apprentissage supervisé, le vecteur de sortie Y est connu. On sépare alors ces données en deux parties :
  - un jeu dit d'entraînement (très souvent 80% de l'ensemble des données),
  - un jeu de test (20% restants).
- La deuxième phase est appelée apprentissage ou entraînement permet de définir le modèle prédictif à partir du jeu d'entraînement (méthode souvent itérative).
- La troisième phase est appelée inférence. Pendant cette phase, on utilise le modèle précédemment défini pour prédire la sortie sur le jeu de test et comparer à la sortie connue. Cette comparaison permet d'analyser les performances de l'algorithme.

II.2. Phase d'apprentissage.

La plupart des algorithmes d'apprentissage fonctionnent en trois temps :

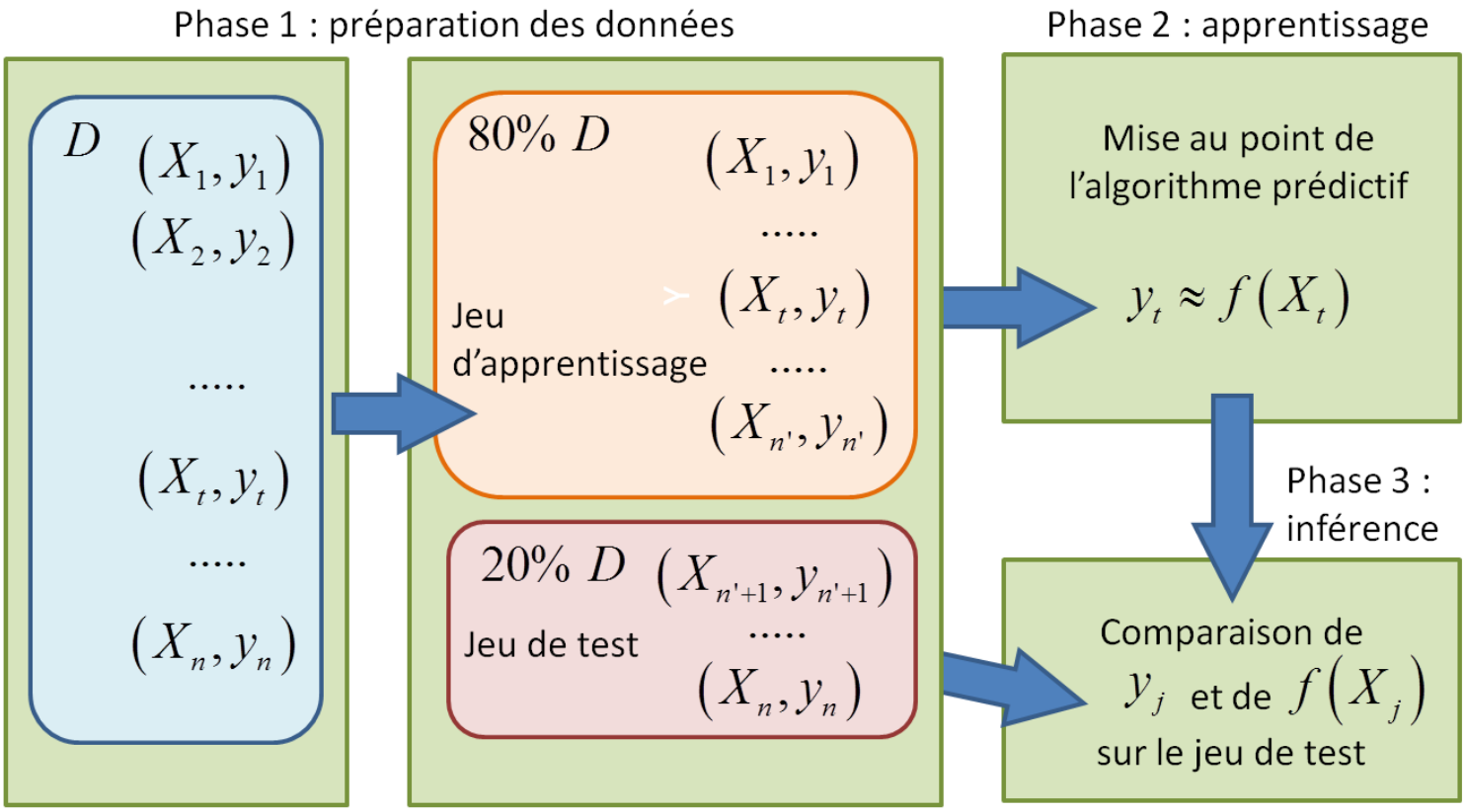
- Choix d'un modèle et d'hypothèses simplificatrices,
- Construction d'une fonction coût,
- Minimisation de cette fonction coût.

II.3. Phase d'inférence.

La qualité d'un modèle est mesurée en comparant sur le jeu de test, la sortie connue et la prédiction proposée par le modèle mis au point pendant la phase d'apprentissage.
Les mesures de cette performance dépendent de la nature des sorties :

- Sorties discrètes (cas de classification), on utilise alors la justesse (Accuracy). Il s'agit de la proportion de prédiction égale à la sortie : $accuracy = \frac{V}{N}$ avec V le nombre de valeurs correctement prédites et N le nombre de prédictions.

- Lorsque les sorties prennent des valeurs continues (régression), on utilise l'erreur moyenne quadratique (Mean Squared Error) : $MSE = \frac{1}{N}\sum_{t=1}^{N}\left(Y_t - \tilde{Y}_t\right)^2$ où Y_t représente la sortie connue et $\tilde{Y}_t$ la prédiction.



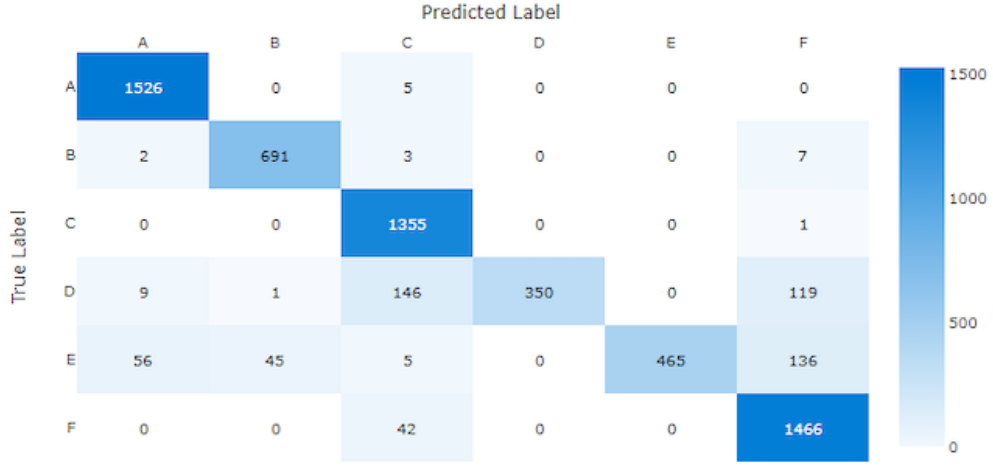
II.4. Matrice de confusion.

Dans le cas d'un problème de classification binaire (2 valeurs pour la sortie), on peut représenter les résultats de la performance de l'algorithme sous la forme de la matrice de confusion :

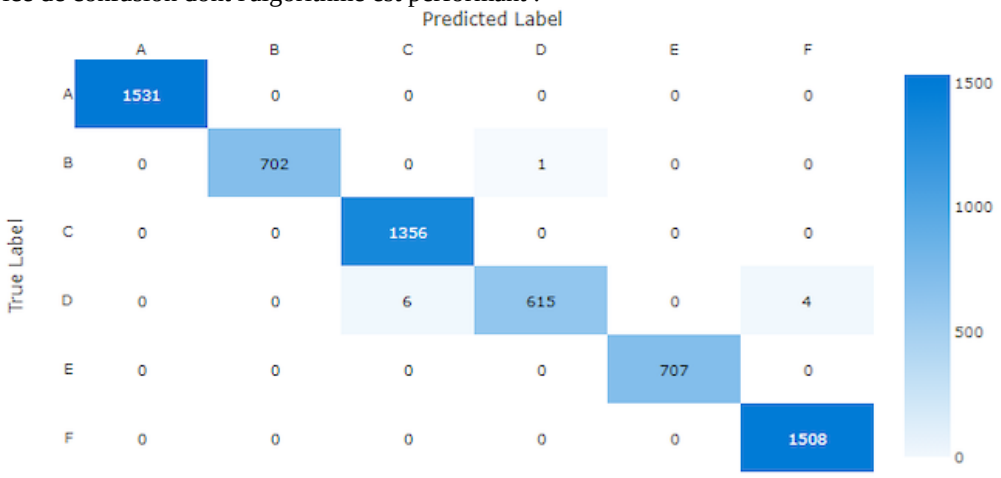
	Vrai prédit	Faux prédit
Vrai réel	100	10
Faux réel	20	200

Plus le nombre d'élément sur la diagonale est élevé, plus l'algorithme est performant.

Dans le cas où la sortie peut prendre p valeurs, on retrouve une matrice de confusion p x p.
Exemple de matrice de confusion 6 x 6 dont l'algorithme n'est pas performant :



Exemple de matrice de confusion dont l'algorithme est performant :



Syntaxe python

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, accuracy_score
```

Si on note : `y_true` et `y_pred`, deux listes contenants respectivement les valeurs de la sortie et sa prédiction par l'alogrithme.

```
accuracy_score(y_true, y_pred)           #justesse
cm = confusion_matrix(y_true, y_pred)    #matrice de confusion
```

Et donc, pour afficher la matrice de confusion :

```
disp = ConfusionMatrixDisplay(confusion_matrix = cm, display_labels = modele.classes_)
disp.plot()
plt.show()
```

III. Principaux algorithmes.

III.1. Algorithmes de régression.

a. Régression linéaire univariée ou monovariée

Pour une régression linéaire univariée, le nombre de colonne de la matrice X vaut 1, c'est à dire qu'il n'y qu'un seul argument. On cherche alors à exprimer la fonction f sous la forme $aX+b$ (a et b étant des constantes). La fonction coût peut se mettre sous la forme de la distance entre Y (valeurs des sorties) et $\tilde{Y}$ (valeurs des prédictions) :

$$J(a,b)=\sum_t \left(y_t - f(x_t)\right)^2 = \sum_t \left(y_t - ax_t - b\right)^2$$

On cherche à choisir les valeurs de a et b qui minimise la fonction coût $J(a,b)$. Pour cela, on doit résoudre le système :

$$\begin{cases} \dfrac{\partial J(a,b)}{\partial a} = 0 \\ \dfrac{\partial J(a,b)}{\partial b} = 0 \end{cases}$$

Pour résoudre ce système, il existe au moins deux méthodes :

- Méthode de descente du gradient : cette méthode itérative permet de résoudre la minimisation de tout type de problème (la descente du gradient n'est pas au programme des CPGE).
- Méthode des moindres carrés : cette méthode s'applique au système univarié. Elle consiste à résoudre directement le système précédent, ce qui revient à résoudre un problème matriciel.

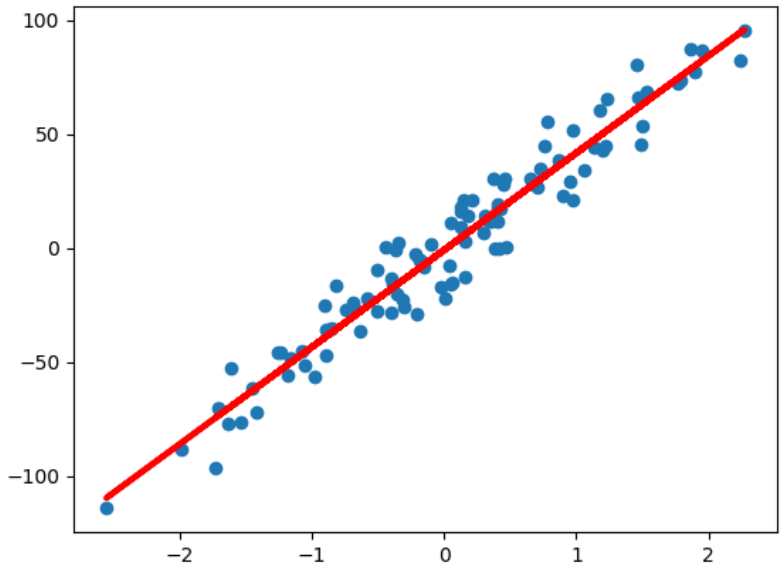
Syntaxe python

Commandes Python :

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

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

#Création du modèle
model = SGDRegressor(max_iter = 1000,
eta0 = 0.001)
model.fit(x,y)
print('Coeff R2 =', model.score(x,y))
plt.plot(x, model.predict(x), c =
'red', lw = 3)
```



b. Régression linéaire multivariée

La régression multivariée est une généralisation de la méthode précédente dans le cas où le nombre d'attributs (colonnes de X) est supérieur à 1. On note n le nombre d'attributs x_{ij} pour un échantillon i donné avec j compris entre 1 et n. et on cherche alors une fonction sous la forme : $\sum_{j=1}^{n} c_j x_{ij} + c_0$.

La fonction coût est définie par $J(c_0, c_1, ..., c_n) = \sum_i \left(y_i - \left(\sum_{j=1}^{n} c_j x_{ij} + c_0 \right) \right)^2$.

On peut alors utiliser une méthode du gradient ou une méthode analytique pour déterminer les coefficients optimaux.

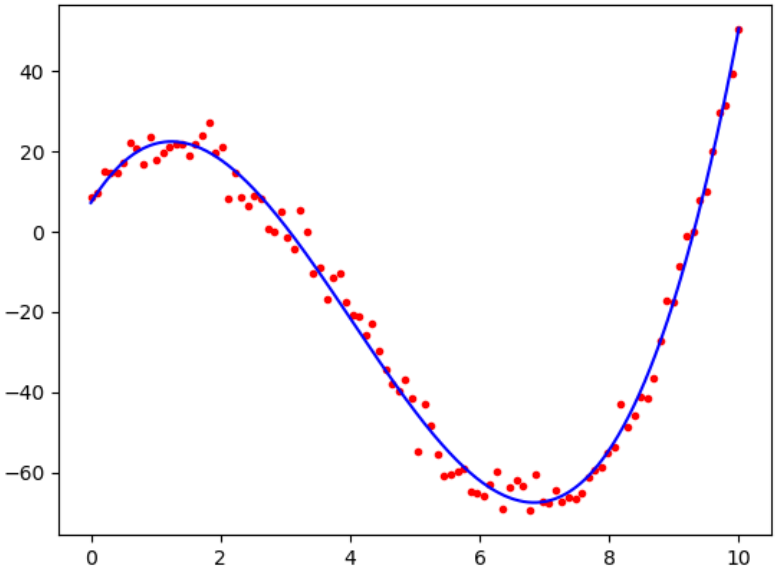
c. Régression polynomiale univariée ou multivariée.

On peut généraliser la méthode précédente à des fonctions polynomiales de degré d si la représentation des données ne semble pas linéaire.
Dans l'exemple ci-contre, on est dans le cas d'une régression univariée polynomiale de degré 3. Le modèle proposé est alors de la forme :

$$f(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

et la fonction coût par la méthode des moindres carrés :

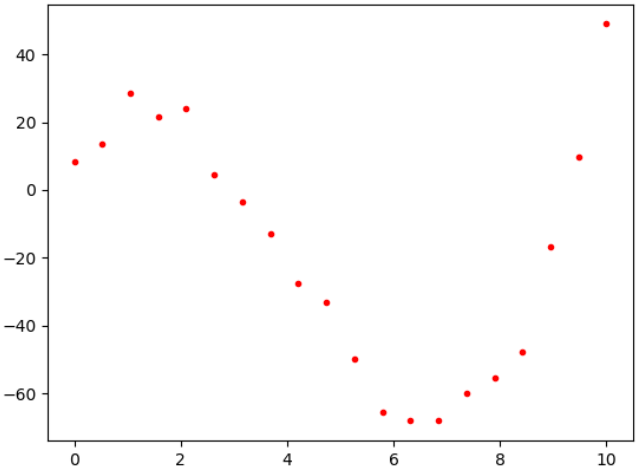
$$J(a_0, a_1, a_2, a_3)$$
$$= \sum_i \left(y_i - \left(a_3 x_i^3 + a_2 x_i^2 + a_1 x_i + a_0 \right) \right)^2$$



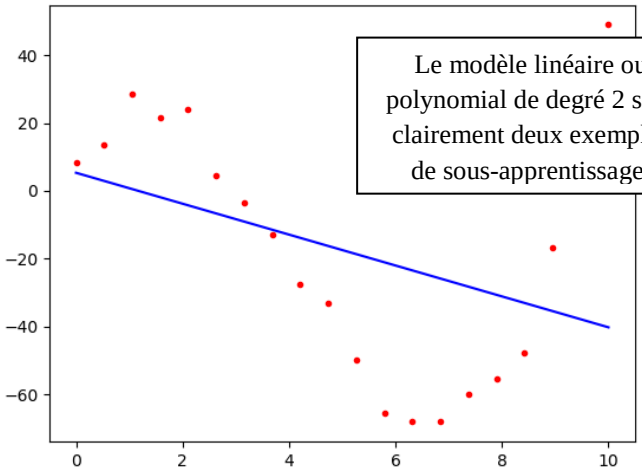
d. Choix de degré pour la régression polynomiale

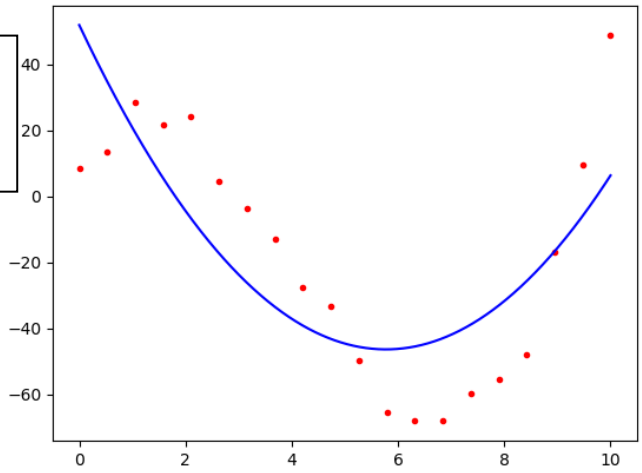
Attention, même si elle peut beaucoup améliorer l'ajustement d'un modèle, la régression polynomiale n'est pas sans risque. En effet, si l'on choisit un degré de polynôme trop grand, on risque de construire une fonction qui passera par un nombre élevé de points des données d'apprentissage, mais selon une forme trop oscillante. Cette fonction modélisera le bruit de mesure et sera mauvaise d'un point de vue prédictif. On parle alors de sur-apprentissage (voir exemple suivant).

Exemple d'un ensemble de $n = 20$ points dont on cherche à définir un modèle.

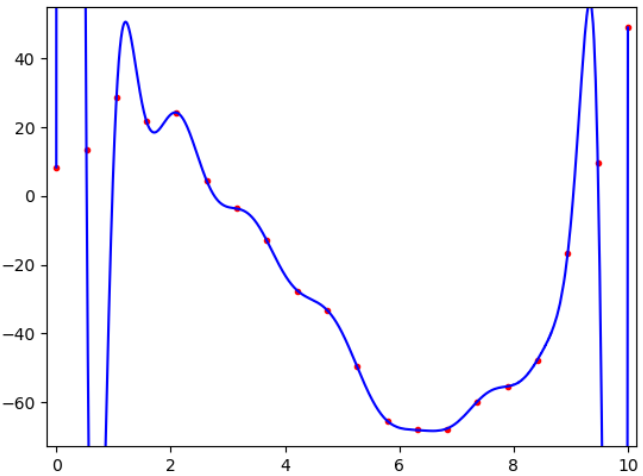


Le modèle linéaire ou polynomial de degré 2 sont clairement deux exemples de sous-apprentissage.

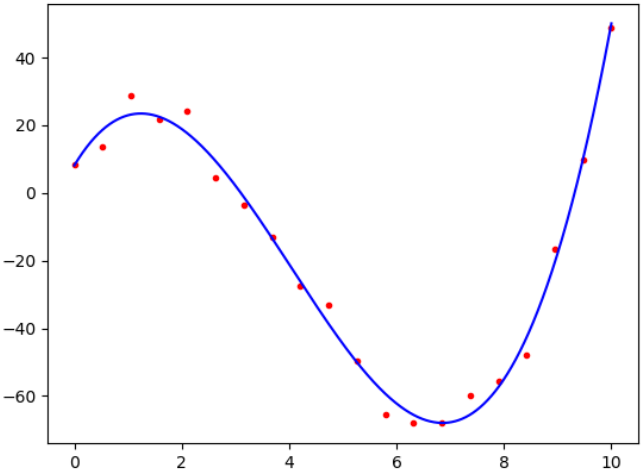




Le modèle polynomial de degré $n-1$ permet de définir une courbe passant par tous les points et est un cas de sur-apprentissage.



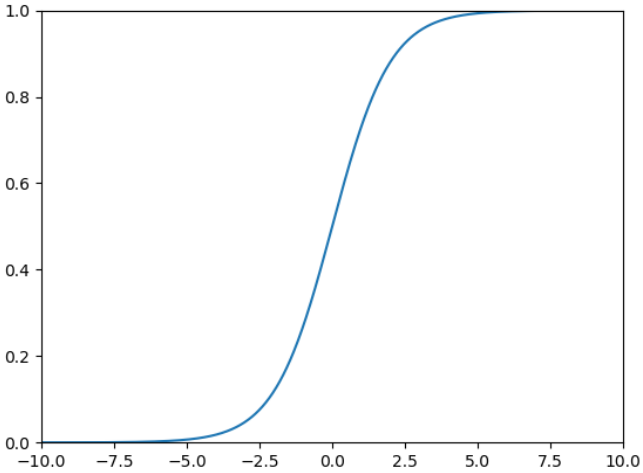
Le modèle de degré 3 est le plus satisfaisant.



III.2. Algorithmes de classification : régression logistique

L'algorithme principal que nous allons étudier est la Régression logistique (contrairement à son nom, il s'agit bien de classification).
Ce modèle permet de classer les données en deux catégories 0 et 1 (sortie binaire). Le principe est exactement que pour la régression linéaire. On définit une hypothèse sur la forme de la relation entre les entrées et la sortie binaire et une fonction coût que l'on minimise pour déterminer les meilleurs paramètres de la forme proposée.
Il n'est évidement par pertinent de choisir une fonction coût linéaire pour prédire une valeur binaire.

La plupart du temps, on utilise une fonction sigmoïde définie par $\sigma(x) = \frac{1}{1+e^{-x}}$ pour définir cette fonction coût. Cette fonction va modéliser la probabilité d'appartenir à la classe 1.



Tracé de la fonction sigmoïde

Ainsi, disposant de n attributs et m échantillons, le problème consiste à chercher le meilleur jeu de paramètres c_j qui minimise la fonction : $J(c_0,....,c_n) = \frac{1}{1+e^{-\sum_{j=0}^{n} c_j x_j}}$.
Il faut ensuite définir un seuil permettant de classer les données en fonction de leur position par rapport à la régression et ainsi pouvoir répartir en 2 catégories les données.
L'utilisation de la régression logistique en Python se fait de manière suivante en notant `X_train` la matrice des données d'entraînement et `y_train` le vecteur des sorties binaires associées pour chaque échantillon d'entraînement.

Syntaxe python

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression()
clf.fit(X_train, Y_train)
y_pred = clf.predict(X_test)
```

III.3. Algorithmes des k plus proches voisins k-NN.

a. Principe.

L'algorithme (supervisé) des k plus proches voisins k-NN (k-nearest-neighbors) consiste à attribuer une valeur prédictive en fonction des valeurs des éléments voisins. On cherche à prédire la sortie y associée à l'entrée x, l'algorithme observe la valeur y_i des k entrées x_i voisins de x.

- pour un algorithme de régression, on prendra la moyenne (ou médiane) des valeurs des éléments voisins,
- pour un algorithme de classification, on prendra la valeur du groupe majoritaire.

Pour faire fonctionner cet algorithme, on a besoin de :

- un ensemble de données X (matrice) et le vecteur de sortie associé Y,
- une fonction distance d,
- un entier k.

Alors l'algorithme de k-NN va :

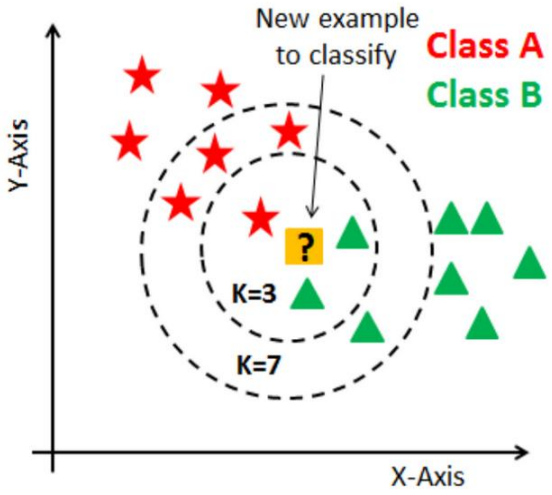
- calculer les distances entre les différents éléments du jeu X et l'élément dont on veut prédire la sortie x : $d(X,x)$,

- retenir le k éléments du jeu dont la distance est la plus petite,
- prendre les valeurs Y pour les k éléments sélectionnés,
- renvoyer :
  - la moyenne (ou médiane) des valeurs Y pour les k éléments sélectionnés pour un problème de régression,
  - la valeur majoritaire des valeurs Y pour les k éléments sélectionnés pour un problème de classification,

b. Distance.

On utilise généralement la distance euclidienne définie par la relation : $D_e(x,y)=\sqrt{\sum_{j=1}^{n}\left(x_j-y_j\right)^2}$ Cette définition est particulièrement adaptée aux données qualitatives (poids, salaires, tailles, montant,...) et du même type.
Il existe d'autres types de distances : Manhattan qui est utile pour des données de type différent (âge, taille,...), Hamming,...
Ces calculs de distances ne seront pas à coder car la bibliothèque Scikit Learn du langage Python calcule automatiquement ces distances.

Illustration ci-contre : Suivant la valeur de k, la nouvelle entrée à classer peut être affectée à une clase différente.



c. Choix de la valeur de k.

Le choix de la valeur k à utiliser pour effectuer une prédiction avec k-NN varie en fonction du jeu de données :

- Moins on utilise de voisins, c'est-à-dire plus la valeur de k est faible, plus on risque un cas de sous-apprentissage.
- Plus on utilise de voisins, c'est-à-dire plus la valeur de k est grande, plus la prédiction sera fiable. Mais si la valeur de k s'approche de n, on risque d'avoir du sur-apprentissage et un modèle qui représentera le phénomène, mais aussi son bruit. Et donc dans ce cas, une prédiction qui devient mauvaise.

Le principal défaut de l'algorithme k-NN est qu'il ne propose pas un modèle qui "remplace" les données, mais doit conserver l'ensemble des données et est donc gourmant en stockage numérique.

Syntaxe python

```
from sklearn.neighbors import KNeighborsClassifier, KneighborsRegressor
model = KNeighborsClassifier(k)     # ou KneighborsRegressor(k), si k est non
                                    # précisé, k = 5 (valeur par défaut)
model.fit(X,y)                      # apprentissage
print(model.score(X,y))             # calcul de la justesse sur toutes les données
print(model.predict(test))          # renvoie une valeur prédite pour une donnée test
```

III.4. Réseaux de neurones.

N.B. : ce chapitre est largement tiré du livre « Sciences Industrielles de l'Ingénieur », de Violeau, Crespel et Caignot, Ed. Vuibert)

Les réseaux de neurones reprennent les principes définis précédemment (minimisation de fonction coût) mais introduisent des notions supplémentaires quant à leur structure.

Ils permettent de traiter des problèmes de classification et de régression de manière supervisée (ou non supervisée mais ces réseaux ne seront pas présentés ici).

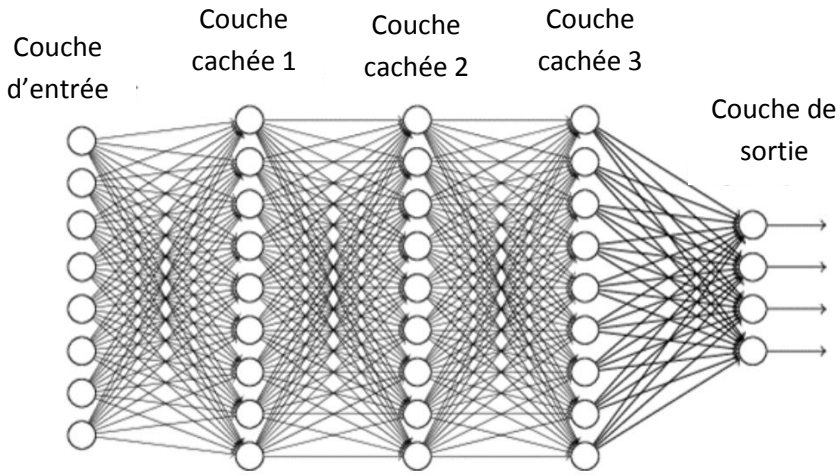
On prend comme exemple un réseau de neurones utilisé pour prédire les propriétés mécaniques d'alliages de magnésium (par exemple limite d'élasticité, dureté ou taille des grains) en fonction de la vitesse de refroidissement et de la composition chimique. Si on ne considère que 2 éléments possibles pour la composition chimique (taux de magnésium et taux d'aluminium), on compte alors 3 attributs d'entrée notés x_i. On dispose d'un ensemble de *m* échantillons pour lesquels les propriétés mécaniques ont été mesurées et sont donc connues.

On connaît donc la matrice *X* des données d'entrée et la matrice *Y* des données de sortie (3 colonnes). L'objectif est donc de déterminer automatiquement les propriétés de nouveaux échantillons.

Un réseau de neurones artificiels, ou *Artificial Neural Network* en anglais, est un système informatique matériel et/ou logiciel dont le fonctionnement est calqué sur celui des neurones du cerveau humain.

a.Structure des réseaux de neurones

Un réseau de neurones est constitué d'une ou plusieurs couches de neurones. Les couches sont reliées entre elles mais aussi avec les données d'entrée et les données de sortie.



On distingue la couche d'entrée qui contient les variables considérées (dans notre exemple 3 : vitesse de refroidissement, taux de magnésium et taux d'aluminium. La couche de sortie contient les valeurs prédites (ici on aurait 3 grandeurs à prédire de propriétés mécaniques). Les couches intermédiaires constituées de neurones seront présentées dans la suite.

b. Modèle du perceptron

Le modèle élémentaire d'un neurone d'une couche associée à des données d'entrée est appelé *perceptron.*

Le but est de savoir si le neurone s'active ou non en fonction des données auxquelles il est relié. Le neurone va donc calculer une grandeur notée *z.*

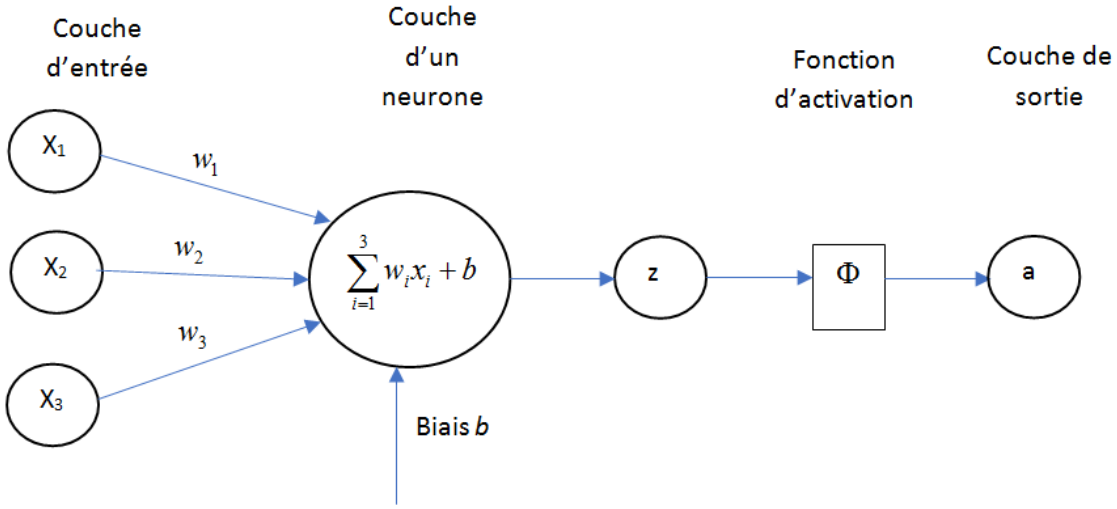
La grandeur *z* fait intervenir des poids w_j devant chaque donnée d'entrée x_j, ces poids sont des valeurs réelles (positives ou négatives). Ainsi un neurone calcule $z = \sum_j w_j x_j$ *On* ajoute une valeur constante à cette somme appelée *biais,* notée *b* : ainsi $z = \sum_j w_j x_j + b$.

Dans l'exemple, la comparaison de *z* à une valeur seuil permettrait d'estimer par exemple si les grains ont une grande taille ou une petite taille (classification binaire). Ainsi si $z \geq 0$ alors la sortie du neurone vaudra **1,** sinon elle vaudra **0** : on dit que le neurone est activé ou non. On comprend ainsi que *b* correspond à la difficulté à activer ou non un neurone.

c. *Fonction d'activation*

L'inconvénient de la grandeur précédente est qu'elle est linéaire et on ne peut prévoir que 2 états pour la variable de sortie. Si on souhaite déterminer une estimation d'une grandeur réelle (limite d'élasticité par exemple), il faut adapter la grandeur calculée par le neurone *z.*

On applique ainsi au neurone une fonction d'activation notée Φ qui va décider si le neurone s'active ou **non.** On calcule alors l'expression $a = \Phi(z)$.

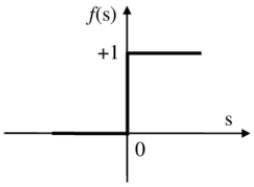


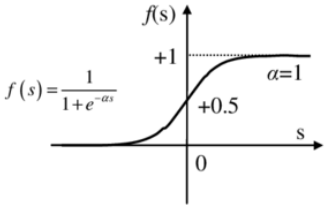
Il existe de nombreuses fonctions d'activation, mais seules trois seront présentées :

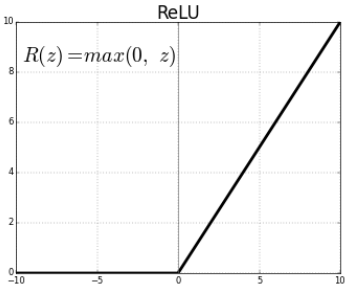
• fonction échelon ou de seuil

• fonction sigmoïde déjà présentée pour la régression logistique

• et fonction ReLu.







La fonction échelon est celle utilisée dans l'exemple précédent de détermination de taille de grains. La fonction sigmoïde (notée *u)* renvoie une valeur continue entre O et 1 ce qui permet de donner plus de liberté sur l'activation d'un neurone (celui-ci sera plus ou moins activé en fonction de la valeur renvoyée).

La fonction Relu est plus utilisée en pratique que la fonction sigmoïde. Dans tous les cas, la fonction d'activation est non-linéaire.

d. *Evaluation des constantes*

Le réseau de neurones va apprendre sur un jeu de données à prédire correctement les sorties. Ce jeu de données d'apprentissage comporte des couples entrée-sortie (x_{ij}, y_i) (*i* représente un échantillon pour lequel la sortie y_i est connue pour des entrées x_{ij} données).

Pour déterminer les meilleurs poids w_j, on introduit une fonction coût que l'on va chercher à minimiser pour que l'estimation faite par le neurone se rapproche de la valeur de sortie connue. Plusieurs fonctions coûts peuvent être utilisées :

• l'erreur quadratique est la plus utilisée pour les problèmes de régression : on cherche à minimiser la fonction :

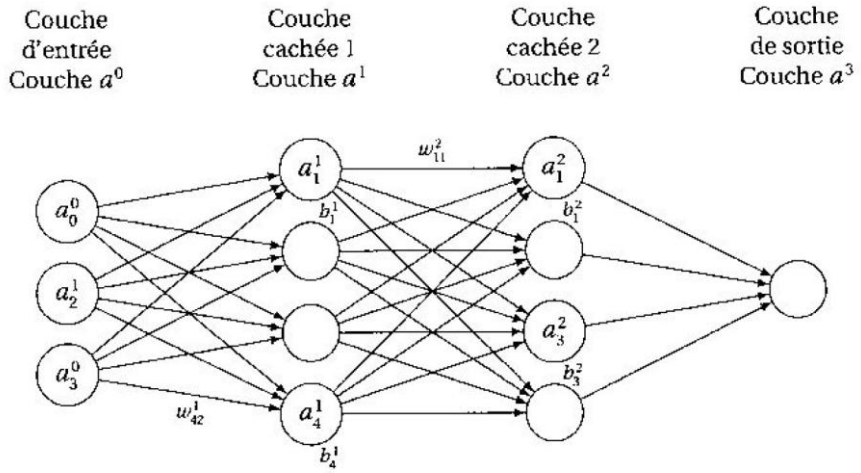
$$C = \sum_i \left| \Phi\left(\sum_j w_j . x_{ij} + b \right) - y_i \right|^2$$

• on utilise plutôt l'entropie croisée *(log-loss)* pour les problèmes de classification : elle ne sera pas détaillée dans le cadre de ce cours.

La méthode utilisée pour déterminer les meilleurs poids et biais est la méthode de descente du gradient. Partant d'un jeu de départ *w1* et *b*, on cherche de manière itérative à partir des dérivées de la fonction *C* par rapport aux poids et biais à converger vers une solution. Cette méthode n'est pas au programme. Cet algorithme dépend d'un paramètre qui spécifie la vitesse de convergence (ou de divergence !) : on parle alors d'hyperparamètre.

e. Réseaux de neurones multicouches

Le modèle du perceptron ne comporte qu'une seule couche constituée d'un seul neurone. Celui-ci ne permet de prédire qu'une seule donnée en sortie. Si on veut pouvoir prédire plus de grandeurs de sortie (dans l'exemple, on souhaite obtenir au moins 3 grandeurs), il faut utiliser plus de neurones et plusieurs couches. On parle de profondeur du réseau de neurones lorsqu'on utilise plusieurs couches. Plus il y a de couches et plus on dit que le réseau est profond *(Deep Learning).*
Dans l'exemple ci-dessous, **le** réseau est composé de 2 couches cachées qui possèdent plusieurs neurones. Le nombre de neurones par couche peut être différent d'une couche à l'autre. C'est un paramètre de la méthode en plus du nombre de couches.
Le principe reste le même qu'avec un perceptron, la différence est que la valeur obtenue par la fonction d'activation sur un neurone est utilisée comme donnée d'entrée pour les couches suivantes.



Remarque
Un réseau qui ne contient pas de couche cachée est appelé réseau simple couche. Il ne contient que la couche d'entrée (qui n'en est pas vraiment une) et la couche de sortie qui contient autant de neurones qu'il y a de grandeurs à estimer.
Le nombre de couches cachées à choisir et le nombre de neurones par couche sont 2 paramètres supplémentaires de la méthode d 'apprentissage automatique. En général on prend 2 couches cachées pour les problèmes simples. Le nombre de neurones par couche peut être quelconque mais plus le nombre est élevé et plus la méthode nécessite de calculs (il y a plus de poids et de biais à déterminer). Il faut en général faire plusieurs simulations jusqu'à trouver un bon compromis entre temps de calcul et justesse du résultat.

Syntaxe python

On utilisera le module `sklearn` (`Scikit-learn`).
Pour traiter un problème de classification, on utilise la classe `MLPClassifier` qui repose sur un réseau de neurones multicouches et qui utilise une méthode de gradient associé à un solveur particulier (ici `lbfgs`).
On peut définir le nombre de couches et de neurones par couche (ici 2 couches respectivement à 5 et 2 neurones). li est possible de spécifier le type de fonction d'activation utilisée (`activation = 'relu'` par défaut ou `activation = 'logistic'` pour une fonction sigmoïde), le maximum d'itérations maximal (`max_iter = 100` par défaut), l'erreur maximale (`tol = 1e-4` par défaut), la possibilité de réutiliser les poids et biais aléatoires initiaux en cas d'appels multiples (`random_state = 1`), etc.
Après définition des données d'entrée X et de sortie Y, le réseau est défini et entraîné par la commande `clf.fit(X,y)`. On peut alors procéder à des prédictions en utilisant la commande `clf.predict(x)` où x est une liste contenant des échantillons tests.

```
from sklearn.neural_network import MLPClassifier
X = [[0., 1.], [1., 2.], [2, 3], [ - 2, - 1]]  # 4 echantillons de 2 attributs
y= [0, 1, 2, 0] # valeurs de sortie à étiquettes
clf = MLPClassifier(solver= 'lbfgs', hidden_layer_sizes=(5, 2), random_state=1)
clf.fit(X, y) #apprentissage
```

```
Yp= clf.predict([[0., 1], [-2., 1]]) #prediction de 2 echantillons
print(Yp)
```

Pour un problème de régression, on utilise la classe MLPRegressor qui fonctionne exactement de la même manière sauf que la fonction d'activation n'est pas appliquée pour la couche de sortie. On peut alors obtenir différentes valeurs pour les données de sortie (et pas uniquement les étiquettes prédéfinies).