Centre d'intérêt 6

Apprentissage Machine

PSI - MP : Lycée Rabelais

1 Introduction au chapitre

L'apprentissage machine (ou *Machine Learning* en anglais) est une branche de "l'intelligence artificielle". L'apprentissage automatique consiste à utiliser beaucoup de données pour apprendre à faire une tâche de manière autonome. Ces données peuvent être des images, des sons, des tableaux, etc.

1.1 Phases d'apprentissage

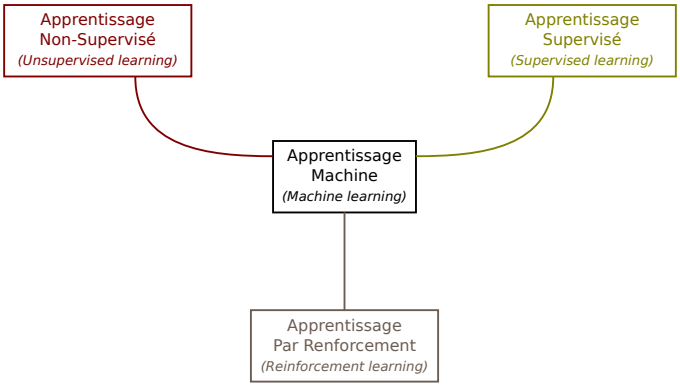
Un algorithme d'apprentissage machine a globalement trois étapes de fonctionnent :

- Une première étape est la **phase d'apprentissage**. On fournit ici des données, dites d'apprentissage, à l'algorithme qui va alors régler ses paramètres internes pour répondre au mieux au problème posé. Cette phase nécessite de très grandes ressources informatiques (serveurs, temps de calcul, données, etc.) afin de déterminer les paramètres du modèle.
- Une deuxième étape est la **phase de test**. On vérifie ici, sur des données particulières, que l'algorithme est bien capable de répondre au problème posé. Si la performance est suffisante, on peut passer à la dernière étape sinon il faut rajouter des données d'apprentissage ou modifier le modèle.
- Dans la dernière étape, dite **phase d'inférence**, on utilise les paramètres déjà établis dans l'étape d'apprentissage pour effectuer de nouvelles prédictions. Cette phase nécessite très peu de ressources informatiques parce que le modèle est établi et n'évolue plus.

1.2 Mode d'apprentissage

D'une manière très générale, il est classique de classer les algorithmes d'apprentissage selon leur mode d'apprentissage. On distinguera (voir schéma ci-dessous) :

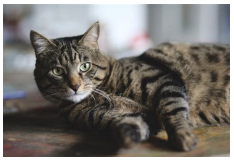
- Les algorithmes **supervisés** qui ont pour but de générer un résultat à partir d'un ensemble de données d'apprentissage où **les entrées et les sorties** de l'algorithme sont connues.
- Les algorithmes **non supervisés** qui ont pour but de générer un résultat à partir d'un ensemble de données d'apprentissage où **les sorties de l'algorithme ne sont pas connues**.
- Les algorithmes par renforcement (hors programme) qui ont pour but de s'améliorer au fur et à mesure de leur utilisation : ici les phases d'apprentissage et d'inférence ne se suivent pas.



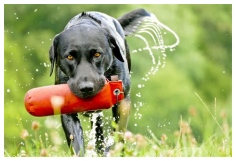
Exemple d'algorithme *supervisé*

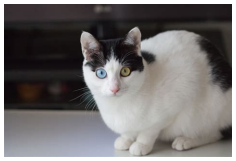
On dispose d'une base de données avec des photos de chats et de chiens. Ces données sont **libellées** : cela signifie qu'une photo de chat est associé à la sortie *chat* et qu'une photo de chien est associé à la sortie *chien*. Un algorithme *supervisé* ajustera ses paramètres afin de classer les images selon les deux groupes prédéfinis *chat* et *chien*.

Exemple de données disponibles pour la phase d'apprentissage :


chat


chien


chien


chat


chien


chat

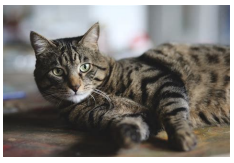
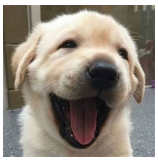

chat

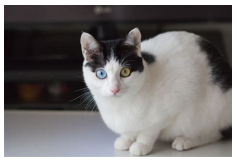

chien

Exemple d'algorithme *non supervisé*

On dispose d'une base de données avec des photos de chats et de chiens **qui ne sont pas libellées**. Dans ce cas, la machine ne sait pas qu'il existe deux groupes *chat* et *chien*. On pourra utiliser un algorithme *non supervisé* afin de classer les images selon deux groupes distincts **non-connus à l'avance**.

Exemple de données disponibles pour la phase d'apprentissage :





Selon certains réglages, on pourra retrouver un regroupement *chat* et *chien* (sans pour autant savoir qu'une photo de chat est associée à un groupe appelé *chat*). On pourra aussi, avec d'autres paramètres, avoir un regroupement en fonction de la couleur, du contraste, etc.

1.3 Régression ou classification ?

Il existe deux grandes classes de problèmes. Les problèmes dits de **régression** et ceux dits de **classification**. Ils se différentient par le type de la sortie attendue.

Pour un problème de **régression**, la sortie peut prendre une infinité de valeurs dans un intervalle.

Pour un problème de **classification**, la sortie ne prendra qu'un nombre fini de valeurs. Lorsqu'il n'y a que deux valeurs de sortie, on parle de classification binaire. Lorsqu'il y a plusieurs valeurs, on parle de classification multiclasse.

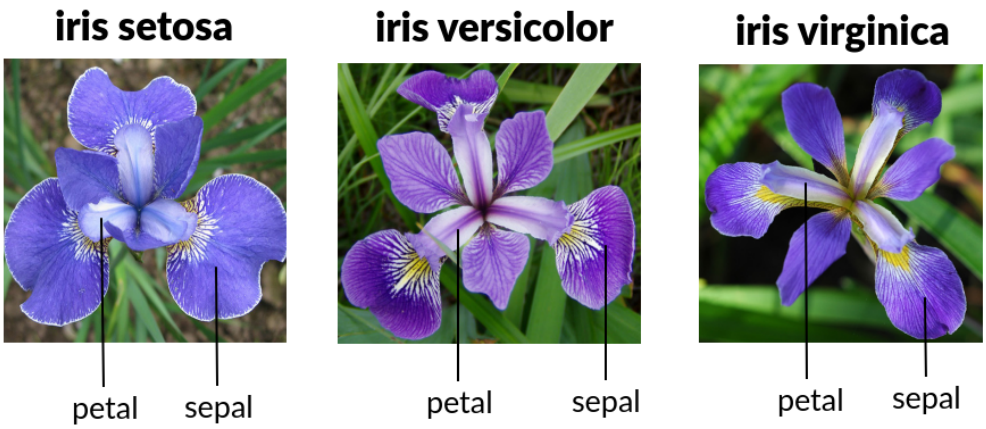
Quelques exemples...

Type d'algorithme : Utilité
..... : Prédire le bénéfice (en euros) d'une production agricole en fonction des données météorologiques, du type de sol, etc.
..... : Prédire l'âge d'une personne à partir d'une photo.
..... : Prédire si un mail est un spam ou non.
..... : Prédire le prix d'un bien immobilier en fonction de sa surface, son emplacement, etc.
..... : Reconnaître la langue d'un texte.
..... : Reconnaître si une plante est comestible ou non.

2 Base de données utilisée pour présenter les algorithmes

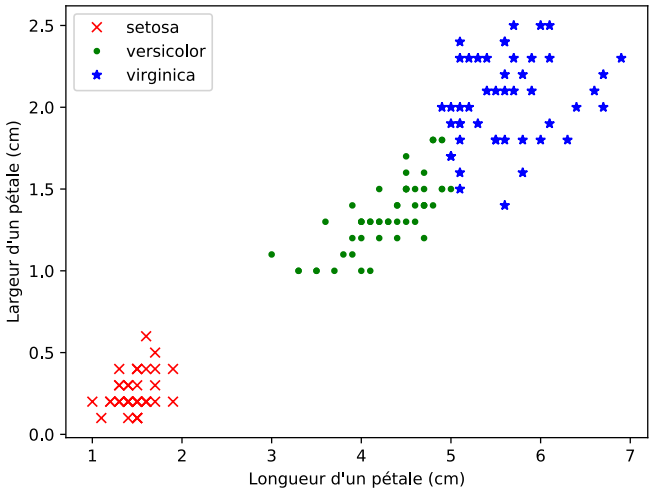
On considère une base de données contenant des mesures sur des fleurs d'iris qui ont été réalisées par des botanistes. Dans cette base de données, on retrouve les longueurs et largeurs des pétales et des sépales pour différentes fleurs et l'espèce d'iris associée : *setosa*, *versicolor* et *virginica*.

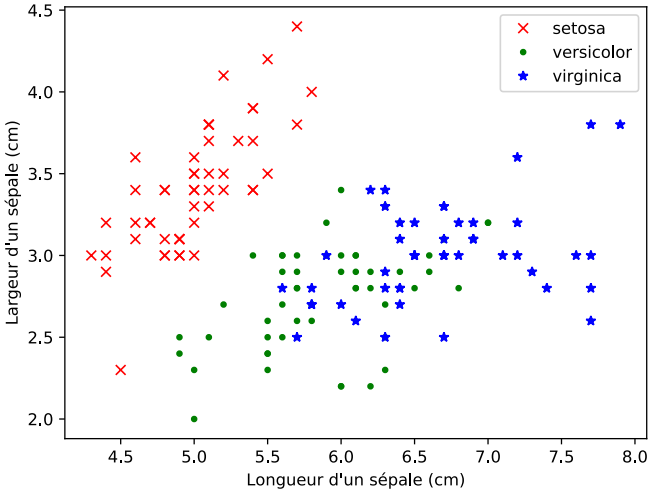
Longueur d'un sépale (cm)	Largeur d'un sépale (cm)	Longueur d'un pétale (cm)	Largeur d'un pétale (cm)	Espèce
5.1	3.5	1.4	0.2	*setosa*
5.5	2.6	4.4	1.2	*versicolor*
6.1	3.0	4.6	1.4	*versicolor*
5.9	3.0	5.1	1.8	*virginica*
...	...	...	...	...



2.1 Problématique de classification

Un premier problème concerne la classification qui permet de prédire l'espèce d'iris (*setosa*, *versicolor* ou *virginica*) en connaissant, par exemple, la longueur et la largeur d'un pétale de fleur.

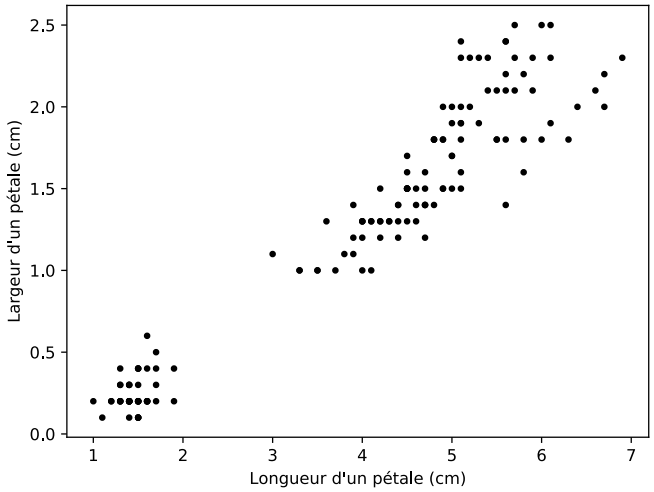




On voit bien sur les tracés précédents que les espèces d'iris peuvent se regrouper en fonction des caractéristiques des pétales ou des sépales.

2.2 Problématique de régression

Un autre problème concerne la prévision de la largeur d'un pétale si l'on connaît la longueur de celui-ci. Ici, c'est bien un problème de régression. On visualise sur le graphe ci-dessous (toutes espèces d'iris confondues) que plus la longueur du pétale est importante, plus la largeur le sera aussi. L'algorithme devra alors trouver une relation entre ces deux paramètres.



2.3 Notations retenues et bibliothèque `Python` utilisée

On notera pour la suite :

- X qui représente les données d'entrée. X sera un tableau de taille $N_{\text{data}} \times N_{\text{carac}}$. N_{data} est le nombre de données d'entrée et N_{carac} le nombre de caractéristiques (ou d'attributs) d'entrée. Ici, par exemple, on aura $N_{\text{data}} = 150$ (150 fleurs ont été répertoriées) et $N_{\text{carac}} = 4$ (longueur et largeur des pétales et sépales ont été mesurées). On pourra parler de la i-ème donnée utilisée que l'on notera x_i. En cas de besoin, on notera $x_{i,j}$ qui correspondra à j-ème caractéristique de la i-ème donnée utilisée.
- Y représente les données de sortie et sera de taille $N_{\text{data}} \times 1$. Y prendra des valeurs continues pour des problèmes de régression et des valeurs finies pour des problèmes de classification. On notera y_i la sortie associée à la i-ème donnée utilisée.

`Python` possède de nombreuses bibliothèques permettant de faire de l'analyse de données. L'une des plus commune

(et que l'on utilisera dans le cours) est la librairie `scikit learn`. Pour importer une `fonction` dans un `module` de la bibliothèque, on pourra écrire : `from sklearn.module import fonction`.

On utilisera également les modules issus de la bibliothèque `numpy`.

Un début de programme est détaillé ci-dessous pour exemple. Il permet seulement de préparer les données à travailler en :

- important les bibliothèques nécessaires ;
- récupérant les données et en les stockant dans les variables X et Y (elles sont pré-implémentées sur `Python` concernant les iris) ;
- préparant une portion des données pour la phase d'apprentissage (entrées `X_train` et sorties `Y_train`) et une portion pour la phase de test (entrées `X_test` et sortie `Y_test`) (ici 33% des données seront utilisées pour le test et 67% pour l'apprentissage).
  On notera dans la suite N_{test}, le nombre de données de test et N_{train}, le nombre de données d'apprentissage où $N_{\text{data}} = N_{\text{test}} + N_{\text{train}}$. Dans notre exemple, $N_{\text{train}} = 100$ et $N_{\text{test}} = 50$.
  L'instruction `shuffle=True` permet de mélanger les données avant la séparation pour éviter d'utiliser une base de données déjà triée ce qui engendre de nombreux problèmes. Dans la base de données des iris; par exemple, les données sont pré-classées par type de fleur. Si on utilise 67% des données pour l'apprentissage, il n'y aura quasiment aucune iris *virginica* car elles sont rangées "à la fin" de la base de données.

```
import numpy as np ## appel de la bibliothèque numpy

# récupération de la base de données
from sklearn import datasets
iris = datasets.load_iris()

# définition des entrées/sorties
Y = iris.target
X = iris.data

# préparation des données d'apprentissage et de test
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, shuffle=True, test_size=0.33)
```

3 Qualité d'un algorithme

Pour analyser la qualité d'un algorithme, il faut utiliser des données de test. Il faut bien comprendre que la phase d'apprentissage est terminée et que les paramètres du modèles sont fixés. On cherche seulement, dans la phase de test, à évaluer si la qualité d'un algorithme est satisfaisante étant donnés les paramètres fixés.

3.1 Problème de régression

Considérons un problème de régression monovariable (une caractéristique d'entrée et une sortie) dont les données d'entrée et de sortie sont notées X et Y respectivement.

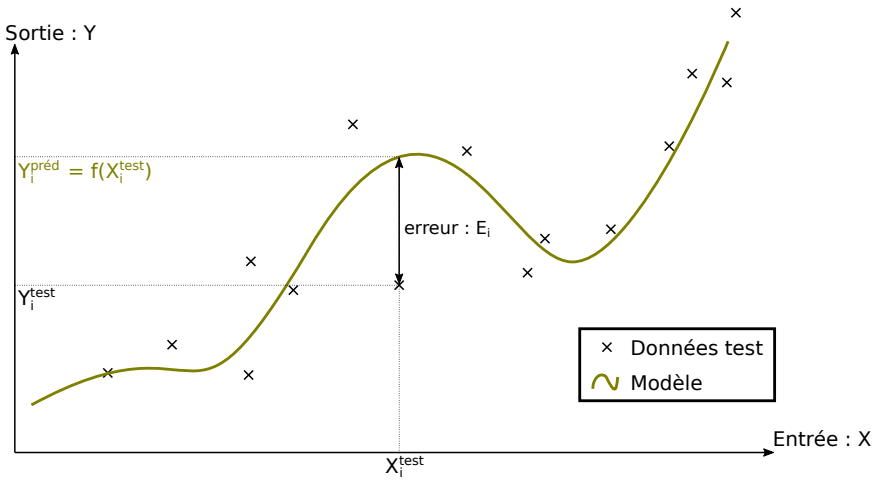
Un **modèle** peut se représenter par une fonction qui permet de prédire une valeur de sortie y pour n'importe quelle entrée x. On a donc simplement $y = f(x)$. Ce modèle a été déterminé avec les données d'apprentissage.

On dispose d'un jeu de N_{test} données de test qui est ici un ensemble de couples $(X_i^{\text{test}}, Y_i^{\text{test}})$. L'indice i correspond à la i-ème donnée.

Pour cette i-ème donnée de test X_i^{test}, on peut aussi prédire la valeur de la sortie avec le modèle en calculant : $Y_i^{\text{préd}} = f(X_i^{\text{test}})$.

Pour évaluer la qualité du modèle, il faut donc comparer $Y_i^{\text{préd}}$ et Y_i^{test}

On peut visualiser cela sur le graphique ci-dessous :



Pour la donnée d'indice i, on peut calculer l'erreur : $E_i = Y_i^{\text{préd}} - Y_i^{\text{test}}$. On veut ensuite calculer la somme des erreurs associées à chaque donnée test. Le problème, si l'on calcule $\sum E_i$, provient du signe de E_i qui peut être tantôt positif et tantôt négatif. De ce fait, la somme des erreurs peut être quasiment nulle alors que les erreurs, en valeur absolue, sont très grandes !

On calculera plutôt, pour la donnée d'indice i, l'erreur quadratique : $(E_i)^2 = (Y_i^{\text{préd}} - Y_i^{\text{test}})^2$.

Enfin, pour avoir un indicateur (on parle aussi de métrique) de la qualité de l'ensemble du modèle avec les N_{test} données de test, on calculera l'erreur quadratique moyenne (*mean squared error*) :

$$MSE = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \left(Y_i^{\text{préd}} - Y_i^{\text{test}} \right)^2$$

On remarquera que l'on ne calcule pas l'erreur globale $\sum \left(Y_i^{\text{préd}} - Y_i^{\text{test}} \right)^2$ parce que ce terme risque d'être très grand, et donc difficilement stockable par un ordinateur, étant donné le nombre très grand de données N_{test} utilisées.

On peut aussi utiliser le coefficient de détermination R^2 défini tel que :

$$R^2 = 1 - \frac{\sum_{i=1}^{N_{\text{test}}} \left(Y_i^{\text{préd}} - Y_i^{\text{test}} \right)^2}{\sum_{i=1}^{N_{\text{test}}} \left(Y_i^{\text{préd}} - \hat{Y}^{\text{test}} \right)^2}$$

Où $\hat{Y}^{\text{test}}$ est la moyenne des valeurs valeurs Y de sortie.

3.2 Problème de classification

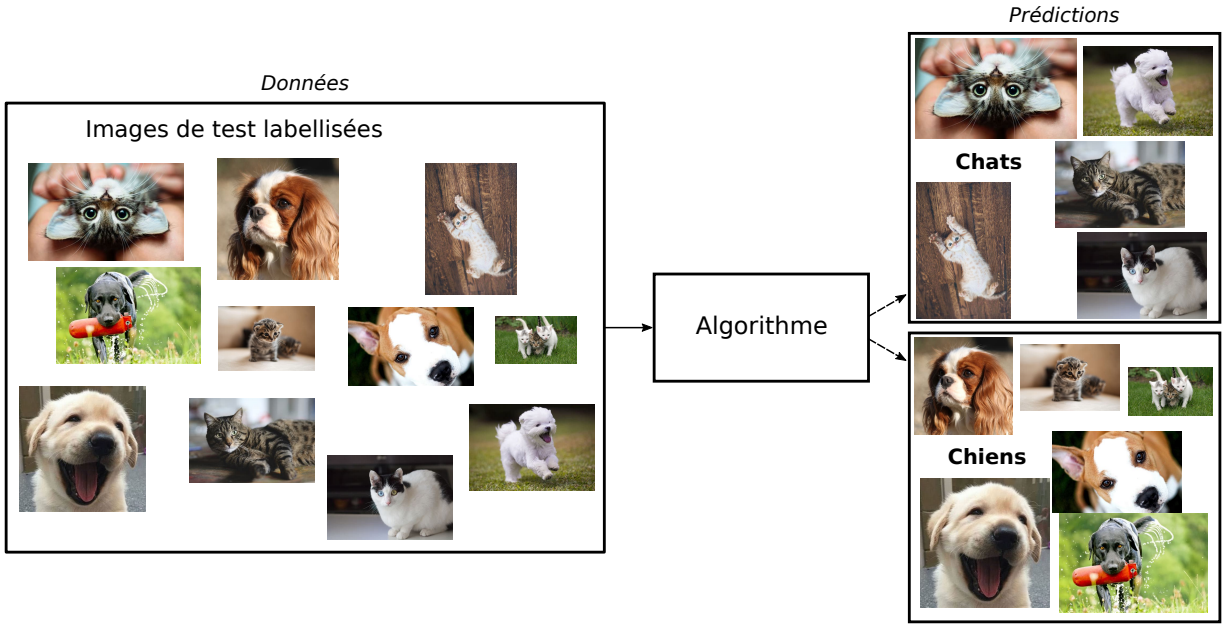
Pour un problème de classification, on utilise généralement **la matrice de confusion** (aussi appelé **tableau de contingence**). Cette matrice est un tableau dans lequel on note :

En ligne : Les comptages associés aux données réelles ;
En colonne : Les comptages associés aux valeurs prédites.

Exemple

On suppose qu'on dispose d'un algorithme qui permet d'identifier, à partir d'une photo, si l'image est un chien ou un chat. Les paramètres de l'algorithme sont déjà déterminés et on cherche simplement à évaluer la qualité de l'algorithme. Pour l'évaluation, on dispose donc d'images labellisées (c'est-à-dire que l'utilisateur sait lorsqu'il s'agit d'un chien ou d'un chat.

Le résultat de l'algorithme (de classification) est donné ci-dessous.



Remplir la matrice de confusion revient à remplir la matrice :

	Chats prédits	**Chiens prédits**
Chats		
Chiens		

On pourra lire le tableau en disant que :

- .... photos de chats ont été utilisées et .... photos de chiens.
- Pour les .... photos de chats utilisées, l'algorithme a donné .... fois la bonne prédiction (un chat) et .... fois la mauvaise (un chien).
- Pour les .... photos de chats prédits, l'algorithme a donné .... fois la bonne prédiction (un chat) et .... fois la mauvaise (un chien).

Un algorithme sera d'autant performant que la matrice de confusions s'apparente à une matrice diagonale.

La **justesse de l'algorithme sera le pourcentage de bonnes prédictions**, c'est-à-dire :

Généralisation

Pour un algorithme de test dont la sortie est binaire : positif ou négatif. Le tableau de contingence s'écrira :

	Sorties positives *(Test positif)*	**Sorties négatives** *(Test négatif)*
Entrées positives *(Personne malade)*	**VP** *(Vrai positif)*	**FN** *(Faux négatif)*
Entrées négatives *(Personne saine)*	**FP** *(Faux positif)*	**VN** *(Vrai négatif)*

On observe que :

- **VP** (vrais positifs) représente le nombre d'entrées positives répondant "Positif" au test,
- **FP** (faux positifs) représente le nombre d'entrées négatives répondant "Positif" au test,
- **FN** (faux négatifs) représente le nombre d'entrées positives répondant "Négatif" au test,
- **VN** (vrais négatifs) représente le nombre d'entrées négatives répondant "Négatif" au test.

On calculera alors :

► la justesse de l'algorithme (*accuracy*). Elle renseigne sur la fiabilité du test. Il s'agit du ratio :

$$\text{justesse} = \frac{N_{\text{bonnes prédictions}}}{N_{\text{test}}} = \frac{\mathbf{VP} + \mathbf{VN}}{\mathbf{VP} + \mathbf{VN} + \mathbf{FP} + \mathbf{FN}}$$

► la sensibilité de l'algorithme. Elle renseigne sur la qualité à détecter une entrée positive. Il s'agit du ratio :

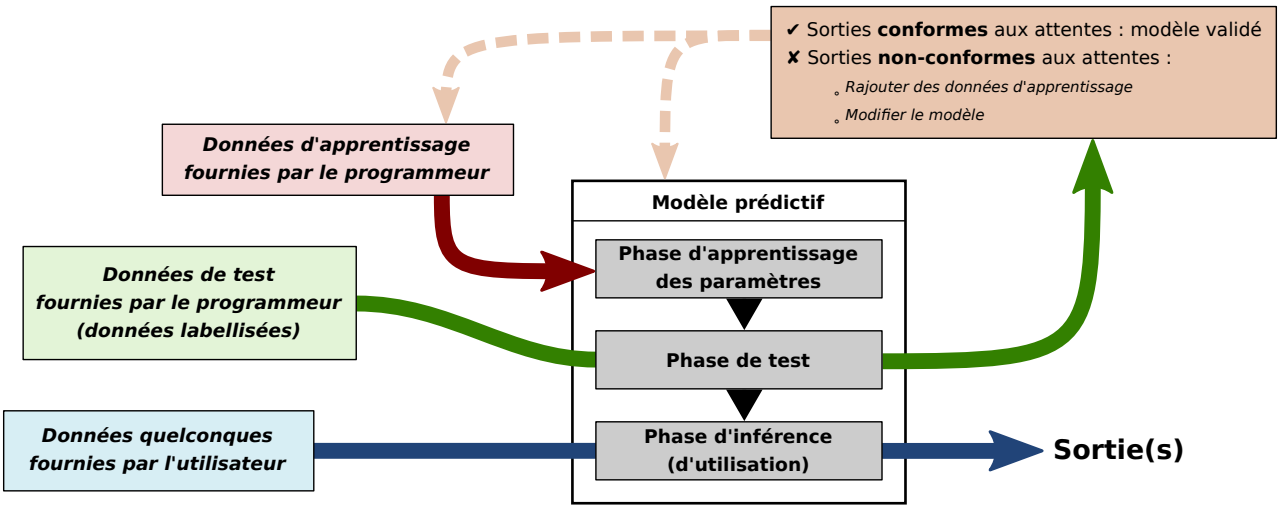
$$\text{sensibilité} = \frac{\mathbf{VP}}{\mathbf{VP} + \mathbf{FN}}$$

► la spécificité de l'algorithme. Elle renseigne sur la qualité à détecter une entrée négative. Il s'agit du ratio :

$$\text{spécificité} = \frac{\mathbf{VN}}{\mathbf{VN} + \mathbf{FP}}$$

3.3 Retour sur la structure d'un algorithme d'apprentissage

Le schéma ci-dessous représente les trois phases de vie de l'algorithme avec, pour chacune, les données à apporter.

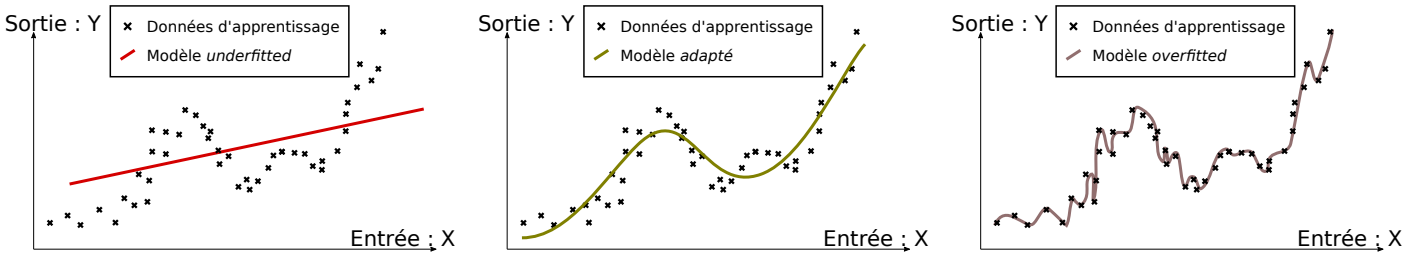


Dans le cas où la phase de test n'est pas satisfaisante, il y a globalement deux possibilités :

- Rajouter des données d'apprentissage : cela signifie que les données n'étaient pas suffisantes et que, de ce fait, les paramètres du modèle ont été évalués de manière inappropriée.
- Modifier le modèle : cela signifie que la structure choisie n'est pas adaptée et donc que le nombre de paramètres du modèle est également inadapté. Un problème récurrent et associé à ce manque est la notion de données "*underfitted*" ou "*overfitted*".

Lorsque les données sont *underfitted*, on parlera aussi de *sousapprentissage*, cela signifie que le modèle ne possède pas assez de paramètres pour prendre en compte les spécificités des données. Dit autrement, la moyenne des données est trop grossière.

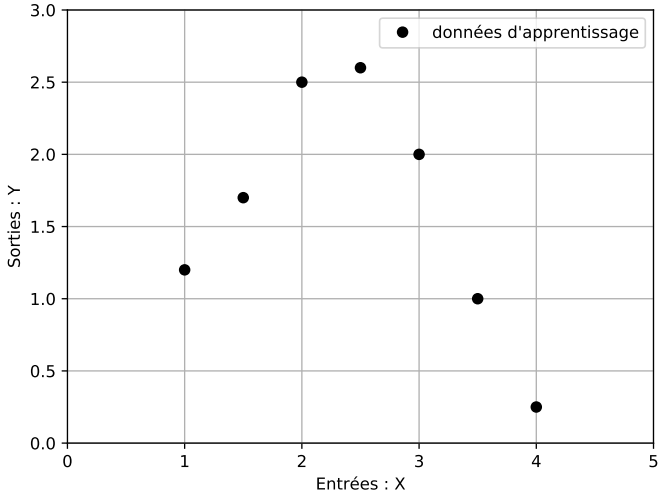
Lorsque les données sont *overfitted*, on parlera aussi de *surapprentissage*, cela signifie que le modèle possède trop de paramètres comparativement au nombre de données d'apprentissage. Dit autrement, le modèle va "suivre" les données d'apprentissage sans en tirer de tendance générale.

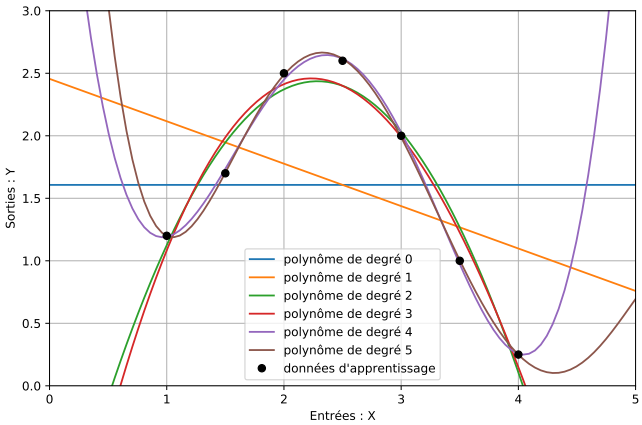


4 Petit exemple avec une régression polynomiale

Considérons un problème de régression monovariable (une caractéristique d'entrée et une sortie) dont les données d'entrée et de sortie sont notées X et Y respectivement.

On cherche à déterminer un modèle de régression polynomiale de telle sorte que $y = f_i(x)$ où f_i est un polynôme de degré i. On dispose des données représentées ci-dessous :



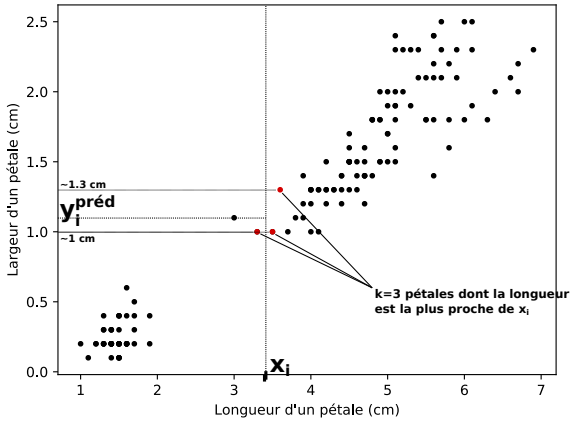


5 Algorithme des k-plus proches voisins

L'algorithme des k-plus proches voisins, aussi noté algorithme *k-NN* pour *k-Nearest Neighbors*, est un algorithme pour l'apprentissage supervisé qui peut être utilisé aussi bien pour des problèmes de régression que pour des problèmes de classification.

5.1 Explication du fonctionnement pour un problème de régression

Explications. On dispose d'un tableau de données d'entrée X correspondant par exemple à des longueurs mesurées sur des pétales. On cherche à prédire les sorties $Y^{\text{préd}}$ correspondant (ici correspondant à la largeur du pétale).



Pour la i-ème donnée de X, il faut suivre les deux étapes de l'algorithme suivant :

1 - Rechercher les k données "voisines" dont l'entrée x de la base des données d'apprentissage est la plus proche de x_i ;

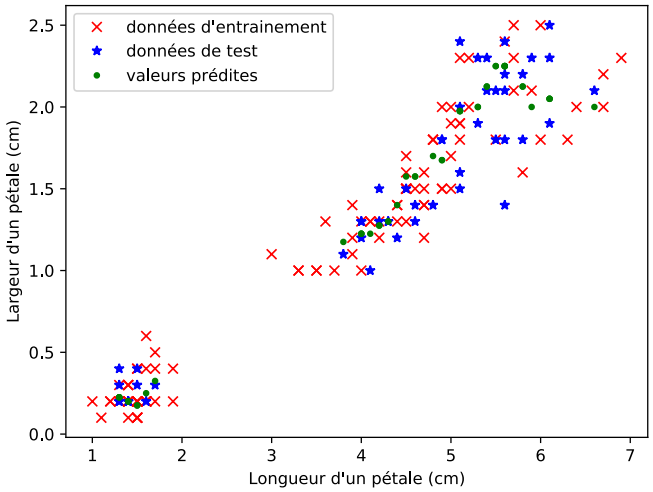
2 - Affecter à $y_i^{\text{préd}}$ la moyenne (ou la médiane) des sorties y de la base de données correspondant aux k plus proches voisins déterminés dans l'étape précédente. Ici, avec $k = 3$, on calcule : $y_i^{\text{préd}} \approx \dfrac{1+1+1.3}{3} \approx 1.1$ cm.

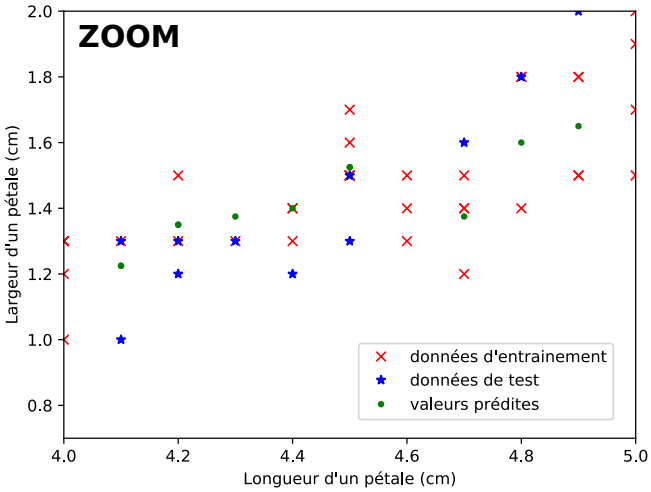
Mise en œuvre sur `Python`.

```python
import numpy as np ## appel de la bibliothèque numpy

# récupération de la base de données
from sklearn import datasets
iris = datasets.load_iris()

# définition des entrées/sorties
X = iris.data[:,2] # longueur du pétale (en cm)
Y = iris.data[:,3] # largeur du pétale (en cm)

# préparation des données d'apprentissage et de test
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, shuffle=True, test_size=0.33)

## pour importer le modèle des k-NN adapté à la régression
from sklearn.neighbors import KNeighborsRegressor
model = KNeighborsRegressor(n_neighbors=3) ## Création du modèle avec le nombre de voisins

model.fit(X_train,Y_train) ## Apprentissage

Y_pred = model.predict(X_test) ## Y_pred est la prédiction du modèle pour les entrées
                               ## test X_test

print('score = ',model.score(X_test, Y_test)) ## Score obtenu

import matplotlib.pyplot as plt
plt.plot(X_train,Y_train,'xr',label="données d'entrainement")
plt.plot(X_test,Y_test,'*b',label="données de test")
```

```
29 plt.plot(X_test,Y_pred,'.g',label="valeurs prédites")
30 plt.legend()
31 plt.xlabel("Longueur d'un pétale (cm)")
32 plt.ylabel("Largeur d'un pétale (cm)")
```

La compilation affiche `score = 0.8955501976801652` et le graphique suivant :

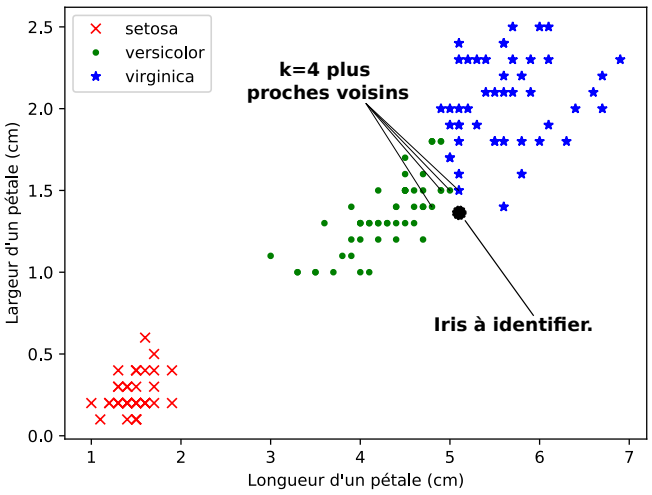




Commenter les résultats obtenus.

5.2 Explication du fonctionnement pour un problème de classification

Explications. On dispose d'un tableau de données d'entrée X. Il correspond, par exemple, à la longueur et à la largeur mesurée sur un pétale et dans ce cas X est un tableau à 2 colonnes. On cherche à prédire le tableau des sorties $Y^{\text{préd}}$ où $Y^{\text{préd}}$ est une valeur finie. Cela correspond dans notre exemple à l'espèce d'iris.



Pour la i-ème donnée de X, il faut suivre les deux étapes de l'algorithme suivant :

1 - Rechercher les k données "voisines" dont l'entrée x de la base de données est la plus proche de x_i ;

2 - Affecter à $y_i^{\text{préd}}$ la valeur du groupe majoritaire. Ici, pour $k = 4$ et l'exemple largeur/longueur, on a 3 iris *versicolor*, 1 iris *virginica* et aucune *setosa*. On prendra donc $Y^{\text{préd}} = \textit{versicolor}$.

Mise en œuvre sur `Python`.

```
import numpy as np ## appel de la bibliothèque numpy

# récupération de la base de données
from sklearn import datasets
iris = datasets.load_iris()

# définition des entrées/sorties
Y = iris.target
X = iris.data[:,2:4] # longueur et largeur du pétale (en cm)

# préparation des données d'apprentissage et de test
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, shuffle=True, test_size=0.33)

## pour importer les modeles des k-NN pour la classification
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=4) ## Création du modèle

model.fit(X_train,Y_train) ## Apprentissage

Y_pred = model.predict(X_test) ## Y_pred est la prédiction du modèle pour les entrées X_test

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_test, Y_pred)
print('score = ',model.score(X_test, Y_test)) ## Score obtenu
print('matrice de confusion =',cm) ## Affichage de la matrice de confusion
```

On donne également la documentation associée à la fonction `confusion_matrix` :

`sklearn.metrics.confusion_matrix(y_true, y_pred)`

Calcule la matrice de confusion pour évaluer la justesse d'une classification.

Paramètres :

- `y_true` : Vecteur de dimension N_{test} contenant les vraies sorties de la base de données
- `y_pred` : Vecteur de dimension N_{test} contenant les sorties prédites par un modèle de classification

Sortie(s) :

- `ndarray` : Tableau qui représente la matrice de confusion dans laquelle la cellule ligne L, colonne C contient le nombre d'éléments de la classe réelle L qui ont été estimés comme appartenant à la classe C

Le résultat affiché est le suivant :

```
score = 0.94

matrice de confusion = [[17 0 0]
                        [0 14 2]
                        [0 1 16]]
```

Commenter les résultats obtenus.

5.3 Calcul de distance

Pour trouver quels sont les k voisins les plus proches, il faudra calculer la distance de l'entrée x_i à toutes les entrés x de la base de données. On retiendra ensuite les k données dont les distance entre x_i et x, notée $d(x_i, x)$, sont les plus faibles.

La plupart du temps, la distance euclidienne est utilisée. En supposant que les données d'entrées possèdent N_{carac} caractéristiques (par exemple $N_{\text{carac}} = 2$ si on s'intéresse en entrée à la largeur et à la longueur d'un pétale), on aura :

$$d(x_i, x) = \sqrt{\sum_{j=1}^{N_{\text{carac}}} (x_{i,j} - x_j)^2} \qquad \text{où } x_{i,j} \text{ et } x_j \text{ sont les } j\text{-ème caractéristiques des données } x_i \text{ et } x.$$

Exemple avec la base de donnée des iris. On donne les mesures suivantes pour une iris rencontrée :

Longueur d'un sépale (cm)	Largeur d'un sépale (cm)	Longueur d'un pétale (cm)	Largeur d'un pétale (cm)
5.3	3.2	1.6	0.4

Remplir le tableau des distances pour les iris de la base de données ci-dessous.

Longueur d'un sépale (cm)	Largeur d'un sépale (cm)	Longueur d'un pétale (cm)	Largeur d'un pétale (cm)	Espèce	d
5.1	3.5	1.4	0.2	*setosa*	
6.1	3.0	4.6	1.4	*versicolor*	
5.9	3.0	5.1	1.8	*virginica*	

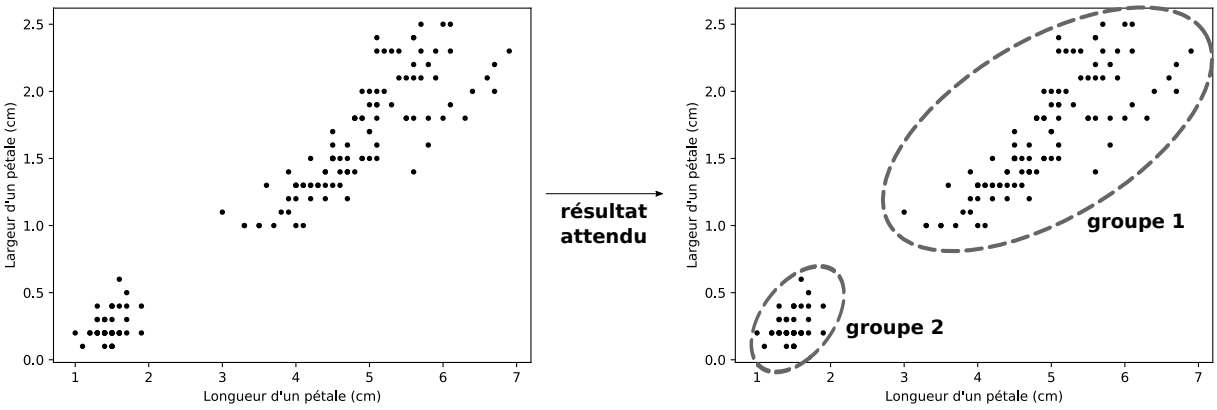
6 Algorithme des k-moyennes

L'algorithme des k-moyennes, aussi noté algorithme *k-means*, est un algorithme pour l'apprentissage non-supervisé qui est utilisé pour des problèmes de classification.

Cet algorithme permet de regrouper les données dans k groupes différents. Cet algorithme n'a donc rien à voir avec celui des k-plus proches voisins !

Explications. On dispose d'un tableau de données d'entrée X correspondant par exemple aux longueurs et largeurs mesurées sur plusieurs pétales (X est donc de taille $N_{\text{train}} \times N_{\text{carac}}$). On cherche à former k groupes distincts. Bien

entendu, ici on suppose qu'on ne connaît pas l'espèce d'iris associée à chaque mesure : il s'agit d'un algorithme à **apprentissage non-supervisé**. Si on choisit $k = 2$, par exemple, l'algorithme aura pour vocation de former deux groupes de la manière suivante :



D'un point de vue mathématique, cet algorithme doit répartir les données en k ensembles $\mathscr{E}_1$, $\mathscr{E}_2$, ... , $\mathscr{E}_k$ de telle sorte que les distances entre les données au sein d'un ensemble soient les plus faibles possibles.

Dit autrement, si on note $X_{\mathscr{E}_j}$ le barycentre des données de l'ensemble $\mathscr{E}_j$, l'algorithme doit répartir les données en k ensembles $\mathscr{E}_1$, $\mathscr{E}_2$, ... , $\mathscr{E}_k$ pour minimiser les distances entre les données d'un ensemble et le barycentre de cet ensemble.

On notera, pour l'ensemble $\mathscr{E}_j$, la somme des distances entre les données de cet ensemble x_i et le barycentre de celui-ci $X_{\mathscr{E}_j}$:

$$S_{ij} = \sum_{i \in \mathscr{E}_j} d(X_{\mathscr{E}_j}, x_i)$$

L'algorithme a donc pour objectif de former k ensembles $\mathscr{E}_1$, $\mathscr{E}_2$, ... , $\mathscr{E}_k$ de telle sorte que la grandeur coût globale :

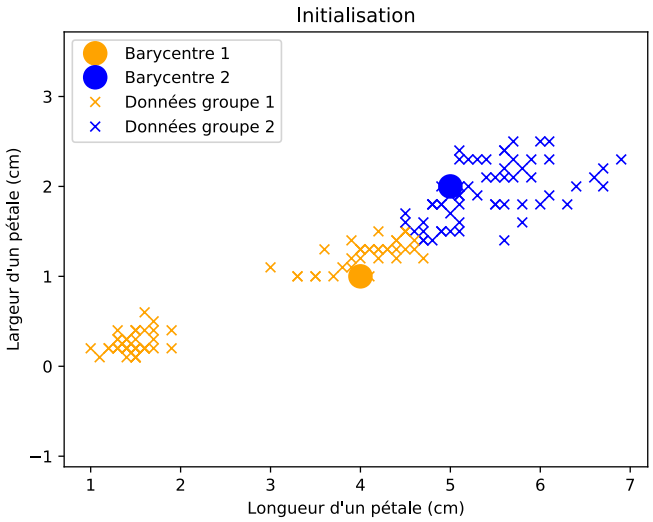
$$\text{coût} = \sum_{j=1}^{k} \sum_{i \in \mathscr{E}_j} d(X_{\mathscr{E}_j}, x_i) \qquad \text{soit minimale.}$$

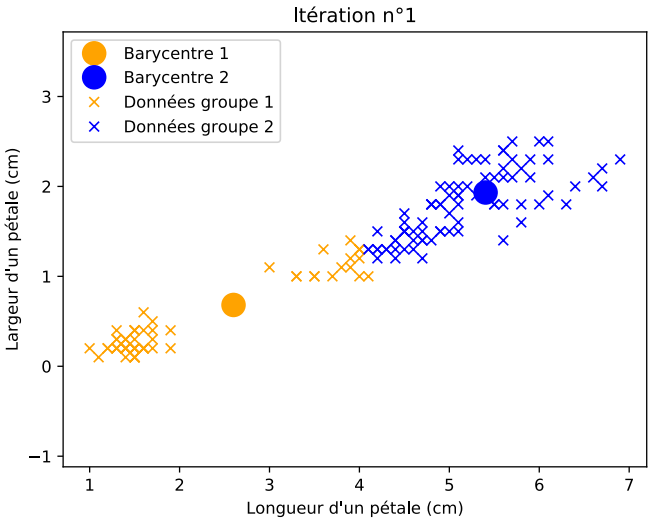
Programmation. D'un point de vue algorithmique, on suivra donc les étapes suivantes pour la phase d'apprentissage :

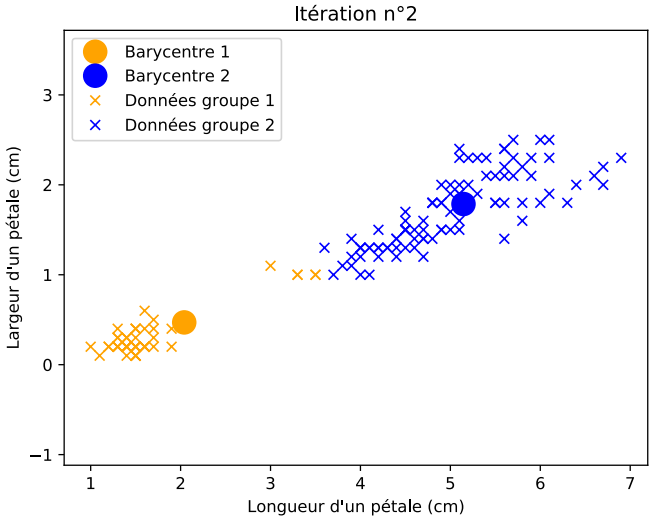
1 - Choisir aléatoirement la position initiale des barycentres $X_{\mathscr{E}_j}$;
2 - Pour chaque donnée, l'affecter au groupe dont elle est le plus proche du barycentre ;
3 - Recalculer la position des k barycentres (ou des k moyennes) pour chaque groupe ;
4 - Réitérer les étapes **2** et **3** jusqu'à ce que la position des barycentres n'évolue plus.

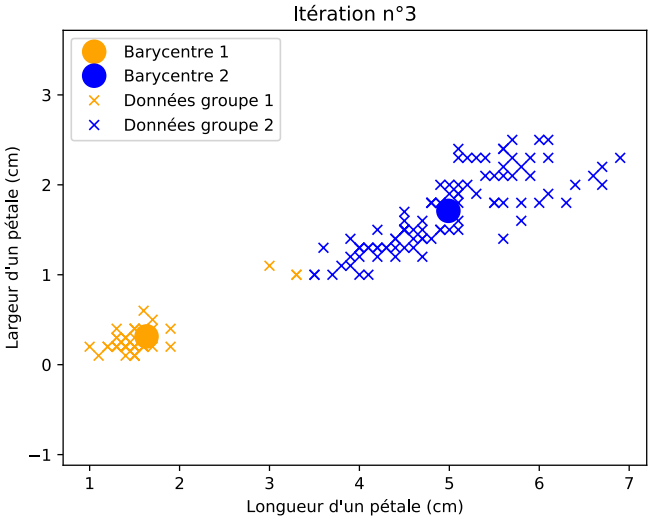
Une fois que le modèle est entrainé, on connait alors la position des k barycentres $X_{\mathscr{E}_1}$, $X_{\mathscr{E}_2}$, ... , $X_{\mathscr{E}_k}$. Pour utiliser le modèle avec une nouvelle donnée, il suffit de regarder de quel barycentre cette donnée est la plus proche et l'affecter au groupe associé.

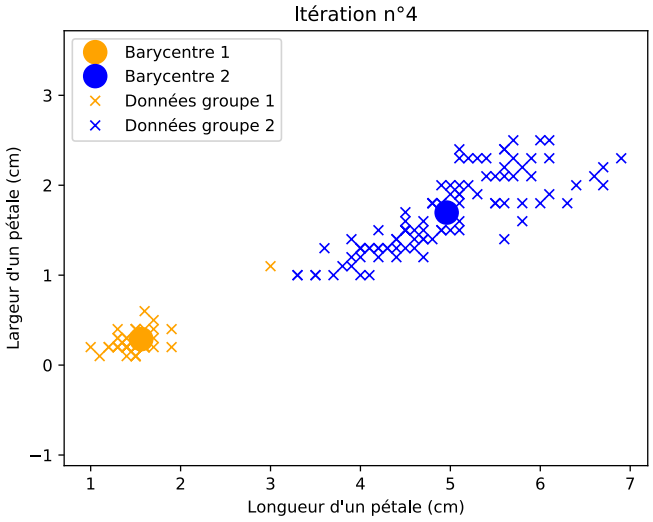
Les graphiques ci-dessous montrent les résultats obtenus en 5 itérations avec une initialisation aléatoire. On a ici décidé un regroupement avec uniquement deux groupes mais l'algorithme fonctionne tout aussi bien pour $k = 3$ ce qui semblerait plus pertinent si l'on sait qu'il y a trois espèces d'iris.

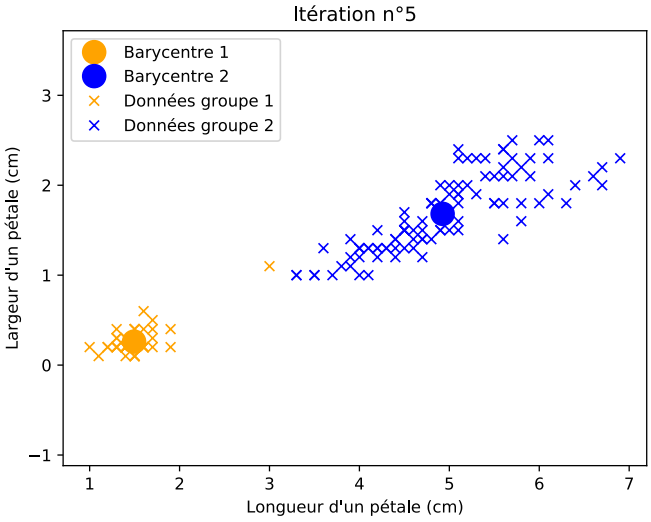












7 Régressions linéaires

7.1 Régression linéaire monovariable

Explications. Ce type de modélisation permet de prédire, pour un tableau de données d'entrée X correspondant par exemple aux longueurs mesurées d'un pétale, la sortie $Y^{\text{préd}}$ correspondant par exemple aux largeurs prédites du pétale dont on connait la longueur. Comme son nom l'indique, la relation recherchée entre l'entrée et la sortie est une relation linéaire. Parler de régression monovariable signifie que l'entrée est à une seule dimension (c'est le cas si X ne représente que la longueur mesurée sur un pétale).

On cherche donc ici a et b tels que $Y^{\text{préd}} = a \cdot X + b$. Durant la phase d'apprentissage, on cherche à minimiser l'erreur quadratique moyenne (ou la distance) entre la prédiction, pour des entrées de la base de données, et la sortie connue de la base de données. Il faut donc chercher a et b pour minimiser la grandeur $J(a,b)$, appelée fonction de coût, définie telle que :

$$\begin{aligned} J(a,b) &= \sqrt{\sum_{i=1}^{N_{\text{train}}} (y_i^{\text{préd}} - y_i)^2} \\ &= \sqrt{\sum_{i=1}^{N_{\text{train}}} (a \cdot x_i + b - y_i)^2} \end{aligned}$$

Chercher a et b pour minimiser la grandeur $J(a,b)$ revient à trouver a et b tels que :

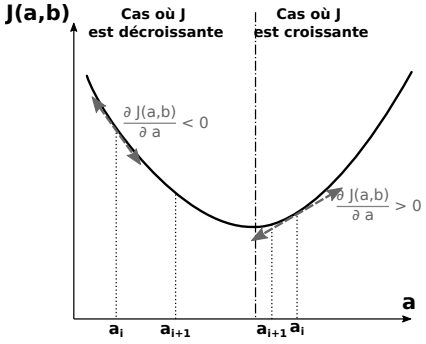
$$\frac{\partial J(a,b)}{\partial a} = 0 \qquad \text{et} \qquad \frac{\partial J(a,b)}{\partial b} = 0$$

Il existe deux méthodes pour résoudre ce type de problème :

- Une première méthode, analogue à celle dite des *moindres carrés*, revient à calculer les dérivées et à résoudre le système d'équation par inversion de matrice (vu en exemple). Cette méthode ne sera pas utilisée car elle présente de nombreux inconvénients liés au nombre de paramètres qui pourra devenir très important lors des régressions multivariables ou dans les réseaux de neurones.
- Une deuxième méthode, dite méthode de la *descente de gradient*, revient à utiliser une méthode itérative. Cette méthode est mieux adaptée avec un nombre important de paramètres. Ce sera donc cette méthode qui sera détaillée puis utilisée dans la suite.

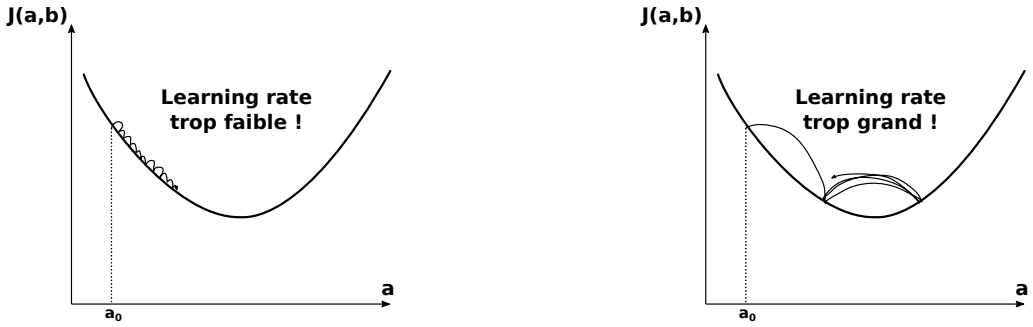
Méthode de la descente de gradient. Pour trouver a et b, il faut observer le graphique ci-dessous et remarquer qu'il y a deux cas à analyser. On notera a_i les valeurs successives de "recherche" de a lors de la descente de gradient.

- Lorsque la fonction de coût J est une fonction décroissante. Pour que les a_i mènent au minimum, il faut qu'ils augmentent. Dit autrement, il faut écrire : $a_{i+1} = a_i + \text{terme positif}$.
- Lorsque la fonction de coût J est une fonction croissante. Pour que les a_i mènent au minimum, il faut qu'ils diminuent. Dit autrement, il faut écrire : $a_{i+1} = a_i + \text{terme négatif}$.



En choisissant $a_{i+1} = a_i - \eta \cdot \dfrac{\partial J(a_i, b_i)}{\partial a}$, on résout le problème du signe à ajouter à a_i : cela permet bien de converger vers le minimum. En choisissant la dérivée, on permet aussi de prendre en compte "l'éloignement vis-à-vis du minimum" : si $J(a_i, b_i)$ est loin de son minimum, il faut donc faire évoluer a rapidement ce qui sera le cas car $\left| \dfrac{\partial J(a_i, b_i)}{\partial a} \right|$ sera grand.

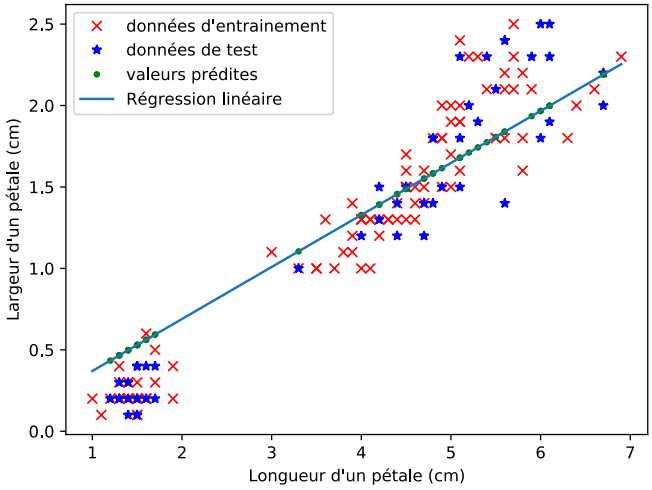
Le paramètre η est appelé vitesse de convergence ou ***learning rate***. On dira que c'est un **hyper-paramètre** du modèle. Si la valeur de η est trop faible, la convergence sera trop lente. Mais si η est trop grand, on peut observer des problèmes de convergence au voisinage du minimum. Ces phénomènes sont représentés sur le schéma ci-dessous. Une valeur classique de *learning rate* est $\eta \approx 0.001$.

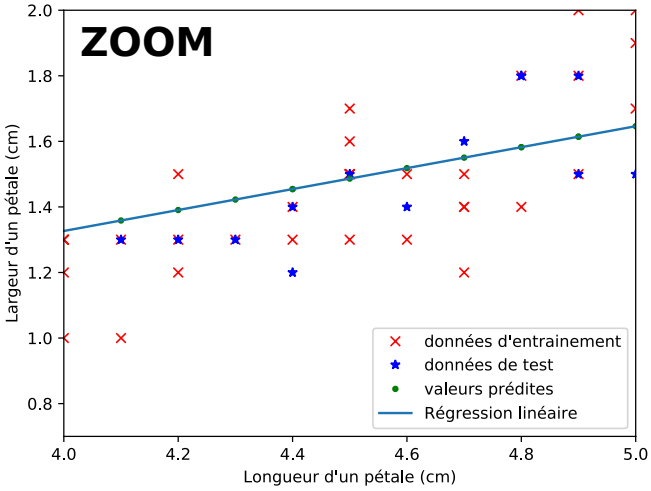


Bien entendu, on a une formule analogue pour b : $b_{i+1} = b_i - \eta \cdot \dfrac{\partial J(a_i, b_i)}{\partial b}$.

Mise en œuvre sur `Python`.

```python
import numpy as np ## appel de la bibliothèque numpy

from sklearn import datasets
iris = datasets.load_iris()

X = iris.data[:,2] # longueur du pétale (en cm)
Y = iris.data[:,3] # largeur du pétale (en cm)

from sklearn.model_selection import train_test_split
## Préparation des données
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,shuffle=True, test_size=0.33)

## pour importer le modèle des k-NN adapté à la classification
from sklearn.linear_model import SGDRegressor
model = SGDRegressor(max_iter=1000,eta0=0.001) ## Création du modèle avec le nombre de voisins

model.fit(X_train,Y_train) ## Apprentissage

Y_pred = model.predict(X_test) ## Y_pred est la prédiction du modèle pour les entrées test
    X_test

print('score = ',model.score(X_test, Y_test)) ## Score obtenu


import matplotlib.pyplot as plt
plt.plot(X_train,Y_train,'xr',label="données d'entrainement")
plt.plot(X_test,Y_test,'*b',label="données de test")
plt.plot(X_test,Y_pred,'.g',label="valeurs prédites")

x = np.linspace(min(X),max(X),2)
y = model.predict(x)

plt.plot(x,y,label='Régression linéaire')
plt.plot()
plt.legend()
plt.xlabel("Longueur d'un pétale (cm)")
plt.ylabel("Largeur d'un pétale (cm)")
```





7.2 Régression linéaire multivariable

Il s'agit ici d'une généralisation de la méthode précédente. Il faut prédire, pour un tableau d'entrée X, le tableau des sorties $Y^{\text{préd}}$. La relation recherchée entre l'entrée et la sortie est une relation linéaire. Parler de régression multivariable signifie que l'entrée a plusieurs dimensions. On peut s'imaginer prédire, par exemple, la largeur d'un sépale en fonction de la longueur et de la largeur mesurées sur un pétale.

Pour la i-ème donnée d'entrée, on cherche donc ici a_1, a_2 ... $a_{N_{\text{carac}}}$ (où N_{carac} est le nombre de caractéristiques de la donnée d'entrée) et b tels que

$$y_i^{\text{préd}} = \sum_{j=1}^{N_{\text{carac}}} a_j \cdot x_{i,j} + b$$

Durant la phase d'apprentissage, on cherche à minimiser l'erreur (ou la distance) entre la prédiction, pour des entrées de la base de données, et la sortie connue de la base de données. Il faut donc chercher a_1, a_2 ... $a_{N_{\text{carac}}}$ et b pour minimiser la grandeur $J(a_1, a_2 ... a_{N_{\text{carac}}}, b)$, appelée fonction de coût, définie telle que :

$$\begin{aligned} J(a,b) &= \sqrt{\sum_{i=1}^{N_{\text{train}}} (y_i^{\text{préd}} - y_i)^2} \\ &= \sqrt{\sum_{i=1}^{N_{\text{train}}} \left(\left[\sum_{j=1}^{N_{\text{carac}}} a_j \cdot x_{i,j} + b \right] - y_i \right)^2} \end{aligned}$$

Où $x_{i,j}$ représente la j-ème caractéristique de la i-ème donnée.

Comme pour la régression linéaire monovariable, on utilise usuellement la méthode de descente de gradient pour évaluer les paramètres a_1, a_2 ... $a_{N_{\text{carac}}}$ et b.

8 Réseau de neurones

Les réseaux de neurones sont basés sur certains concepts vus précédemment. Il permettent de s'adapter à deux nombreux problèmes : apprentissage supervisé ou non-supervisé ; problème de classification ou de régression.

8.1 Structure d'un réseau de neurone

Un réseau de neurones représente un algorithme dont le fonctionnement se rapproche de celui du cerveau humain. Ce réseau de neurones sera constitué de plusieurs couches de neurones.

Chaque neurone génère une sortie à partir de plusieurs entrées.

On distinguera la **couche d'entrée**, les couches dites **cachées**, et la couche de **sortie**. Les neurones seront connectés entre eux : la sortie de l'un pourra être l'entrée d'un ou plusieurs autres.

La couche d'entrée permet simplement de "capter" les entrées à intégrer au modèle. La couche de sortie permet d'exprimer la ou les sorties.

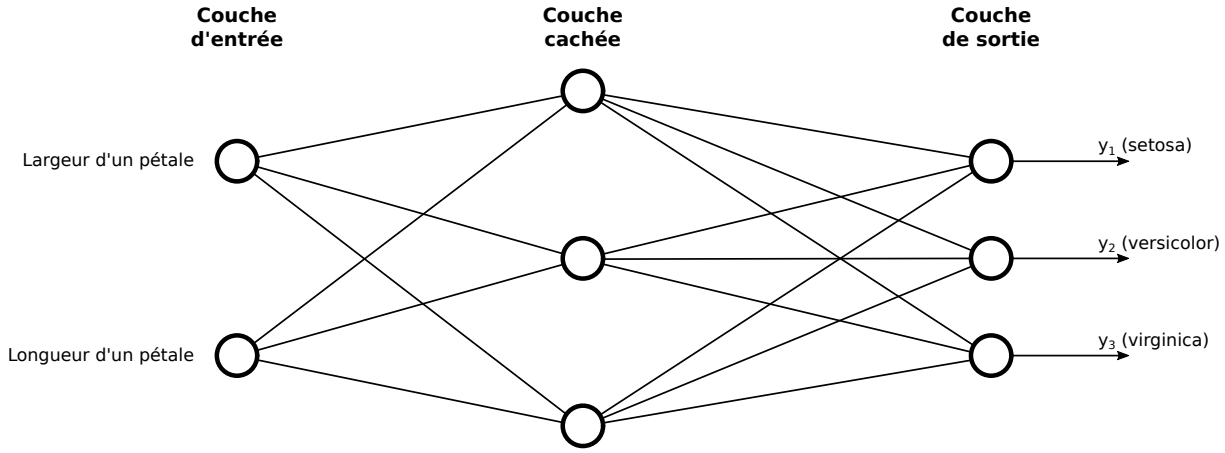
La structure des connexions entre les neurones peut varier d'un modèle à un autre. Nous utiliserons essentiellement des réseaux dits *fully connected* où la sortie d'un neurone d'une couche est connectée à tous les neurones de la couche suivante. Le choix du nombre de neurones dans chaque couche et du nombre de couches est réservé aux spécialistes des data-sciences.

Exemple des iris "en classification"

Si l'on souhaite prédire si une iris est de l'espèce *setosa*, *versicolor* ou *virginica* en fonction de la largeur et de la longueur d'un pétale, on utilisera un réseau avec :

- Deux entrées : une pour la largeur du pétale et une pour sa longueur ;
- Trois sorties : une pour chaque espèce. Chaque sortie pourra, par exemple, être la probabilité d'obtenir l'espèce associée à la sortie. Si le triplet des trois sorties (y_1, y_2, y_3), associées respectivement aux espèces *setosa*, *versicolor* et *virginica*, est $(0.0001, 0.002, 0.987)$, on dira que l'espèce observée est probablement une iris *virginica*.

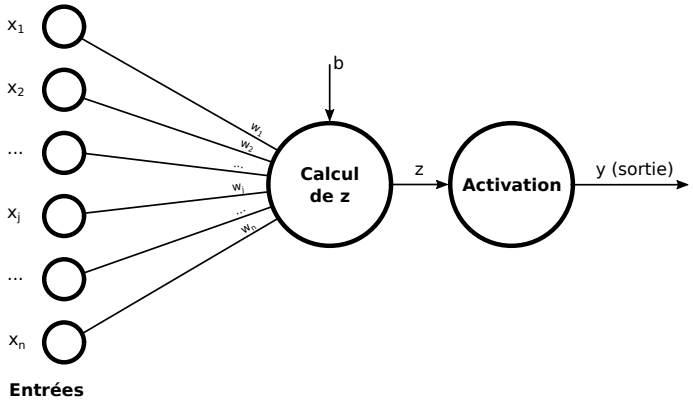
On peut imaginer, par exemple, utiliser une couche cachée à trois neurones. Ceci donnerait donc la structure suivante :



8.2 Fonctionnement d'un neurone

Un neurone fonctionne toujours en deux étapes :

1- Calcul de la grandeur z à partir des **entrées du neurone**, notées x_j.
2- Génération de la sortie à partir de la fonction d'activation $y = \Phi(z)$.



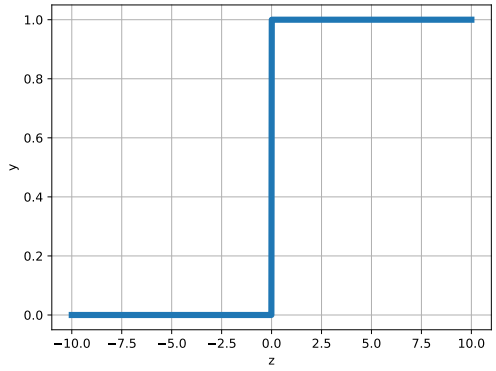
1 - Calcul de z. La grandeur z est une combinaison linéaire des entrées. On exprimera z en fonction :

- des n **entrées** x_j ;
- **des poids** (pondérations) associés à chacune des entrées w_j ;
- **du biais**, noté b, qui est une valeur constante.

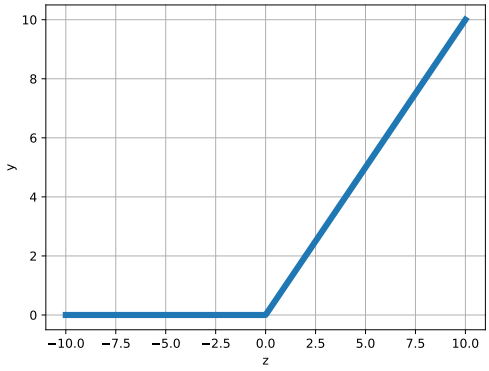
On écrira alors :

$$z = b + \sum_{j=1}^{n} w_j \cdot x_j$$

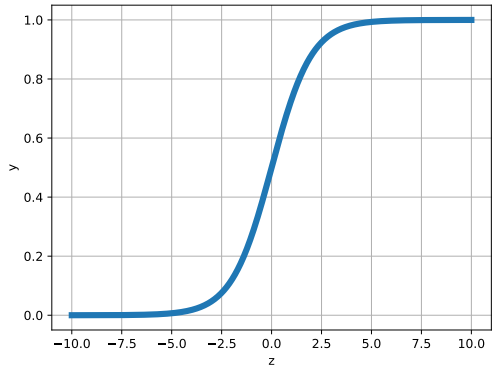
2 - Fonction d'activation. On écrira ensuite $y = \Phi(z)$ où Φ est la fonction d'activation et y la sortie. Il existe de nombreuses fonctions d'activation qui seront à choisir en fonction de la sortie attendue (sortie binaire, sortie comprise dans un intervalle, etc.). Ces fonctions peuvent être non-linéaires ce qui permet une adaptation aisée à de nombreux problèmes. On donne, ci-dessous, l'évolution de quatre fonctions d'activation classiques :



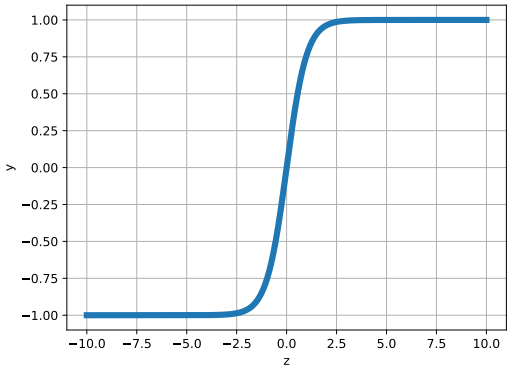
Fonction échelon



Fonction ReLu



Fonction sigmoïde



Fonction tangente hyperbolique

NOTA : ReLu signifie *Rectified Linear Unit* ou, en français, *Unité Linéaire Rectifiée*.

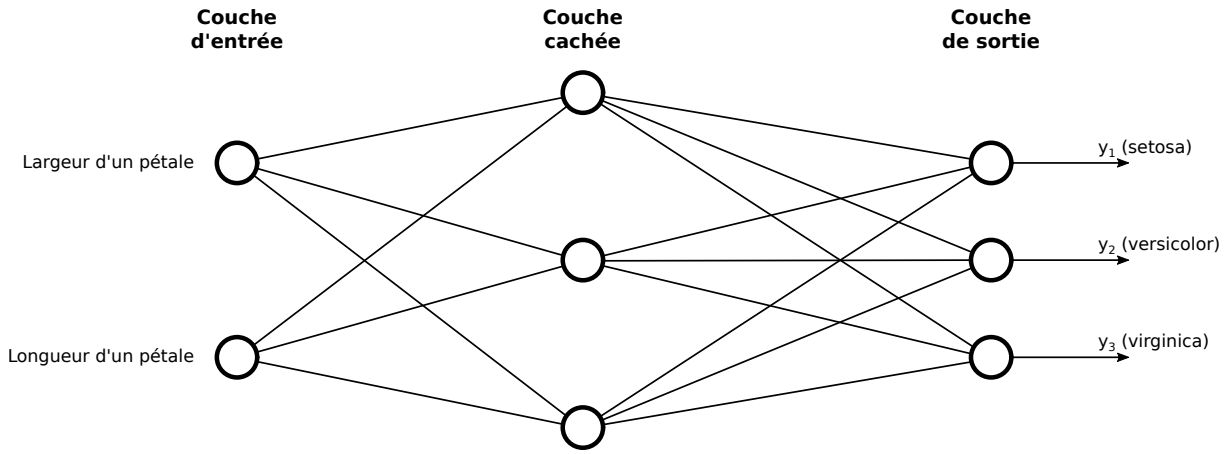
Choix des poids w_j et du biais b. Lors de la création du réseau de neurones, c'est-à-dire au début de la phase d'apprentissage, les poids et les biais de tous les neurones sont choisis aléatoirement. À la fin de la phase d'apprentissage, il faut que ces paramètres permettent de prédire correctement la ou les sorties en fonction de la ou des entrées.

Pour choisir ces paramètres, la méthode est similaire à celle utilisée pour une régression linéaire. Elle se décompose en deux étapes :

- Calcul d'une fonction coût qui dépend de tous les poids et de tous les biais qui apparaissent dans le réseau ;
- Minimisation de cette fonction coût par méthode de descente de gradient.

Exemple des iris "en classification"

Reprenons le réseau proposé précédemment.



Si l'on souhaite que les sorties soient des probabilités de rencontrer l'une ou l'autre des espèces, celle-ci seront donc comprises entre 0 et 1. Choisir comme fonction d'activation la fonction sigmoïde semble donc pertinent.

On peut compter 15 poids et 6 biais. Il faut donc optimiser 21 paramètres pour un problème plutôt simple. Il faut bien comprendre qu'en augmentant le nombre de neurones, le nombre de paramètres à optimiser explose ! Cela rend ces algorithmes très coûteux en temps de calcul durant la phase d'apprentissage et nécessite des serveurs de calculs très performants. C'est aussi cette explosion du nombre de paramètres qui justifie l'utilisation de la méthode de descente de gradient plutôt qu'une méthode par résolution de système linéaire classique par inversion de matrice.

8.3 Utilisation - paramètres et hyperparamètres

Pour l'utilisateur, la résolution des problèmes mathématiques est complètement cachée. L'important, au niveau qui nous concerne, est de savoir créer un modèle puis de régler les hyper-paramètres associés.

Mise en œuvre sur `Python`. On cherche dans le code proposé à créer un modèle permettant d'identifier l'espèce d'iris à partir de la longueur et de la largeur d'un pétale. Rappelons qu'ici, on souhaite que :

- La couche d'entrée possède 2 neurones. Cela est transparent dans la bibliothèque `sklearn` parce que la variable `X` qui est un tableau avec N_{train} lignes (pour l'apprentissage) possède également 2 colonnes associées automatiquement aux deux couches d'entrée.
- La couche de sortie possède 3 neurones. Cela est également transparent à condition que la sortie soit du bon format, à savoir N_{train} lignes (pour l'apprentissage) et 3 colonnes associées à la probabilité d'obtenir telle ou telle espèce. Il faudra donc modifier légèrement le code : pour une iris *virginica*, associée à l'étiquette 2 dans

le tableau d'entrée, il faudra créer la liste `[0,0,1]` (probabilité de 1 d'avoir l'iris de type 2 (rangée à l'indice 2 de la liste)).

Dans les options du modèle associé au réseau de neurones `MLPClassifier` (un réseau de neurones utilisant une optimisation par descente de gradient), on pourra saisir :

- `solver='lbfgs'` : choix du solveur pour l'optimisation des poids et biais.
- `activation='logistic'` : choix des fonctions d'activation (`'logistic'` est la fonction sigmoïde).
- `hidden_layer_sizes=(3)` : choix du nombre de couches cachées et du nombre de neurones associés. Saisir `(4,6,2)` à la place de `(3)` aurait signifier un réseau avec 3 couches cachées : la première avec 4 neurones, la deuxième avec 6 neurones et la dernière avec 2 neurones.
- `random_state=1` : impose un choix aléatoire des poids et biais à l'initialisation.
- `learning_rate_init=0.4` : choix du *learning rate* (ici constant et égal à 0,4).
- `max_iter=200` : choix du nombre maximal d'itération (lors de la descente de gradient).

```
import numpy as np ## appel de la bibliothèque numpy

from sklearn import datasets
iris = datasets.load_iris()

X = iris.data[:,2:] # longueur et largeur du pétale (en cm)
Y = iris.target # type de fleur

from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, shuffle=True, test_size=0.33)



## pour importer le modèle des k-NN adapté à la classification
from sklearn.neural_network import MLPClassifier
model = MLPClassifier(solver='lbfgs',activation='logistic',hidden_layer_sizes=(3),random_state
    =1,learning_rate_init=0.4,max_iter=200) ## Création du modèle avec le nombre de couches et
    de neurones

model.fit(X_train,Y_train) ## Apprentissage

Y_pred = model.predict(X_test) ## Y_pred est la prédiction du modèle pour les entrées test
    X_test

def num(Y):
    res = []
    for j in range(0,len(Y)):
        i = np.argmax(Y[j])
        res.append(i)
    return res

print('score = ',model.score(X_test, Y_test)) ## Score obtenu

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(num(Y_test), num(Y_pred)) ## num(Y) est un tableau de taille N rempli des
     étiquettes 0,1 et 2
print('score = ',model.score(X_test, Y_test)) ## Score obtenu
print('matrice de confusion =',cm) ## Affichage de la matrice de confusion
```

Le résultat affiché est le suivant :

```
score = 0.9
```

```
matrice de confusion = [[16 0 0]
                        [0 18 2]
                        [0 3 11]]
```

Commenter les résultats obtenus.