

Introduction à l'Intelligence Artificielle

1 Un historique de l'approche « intelligence artificielle »

1.1 Définition

Sous le terme intelligence artificielle (IA) on regroupe l'ensemble des **théories et des techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence.** *{Définition de l'encyclopédie Larousse}*

1.2 Histoire

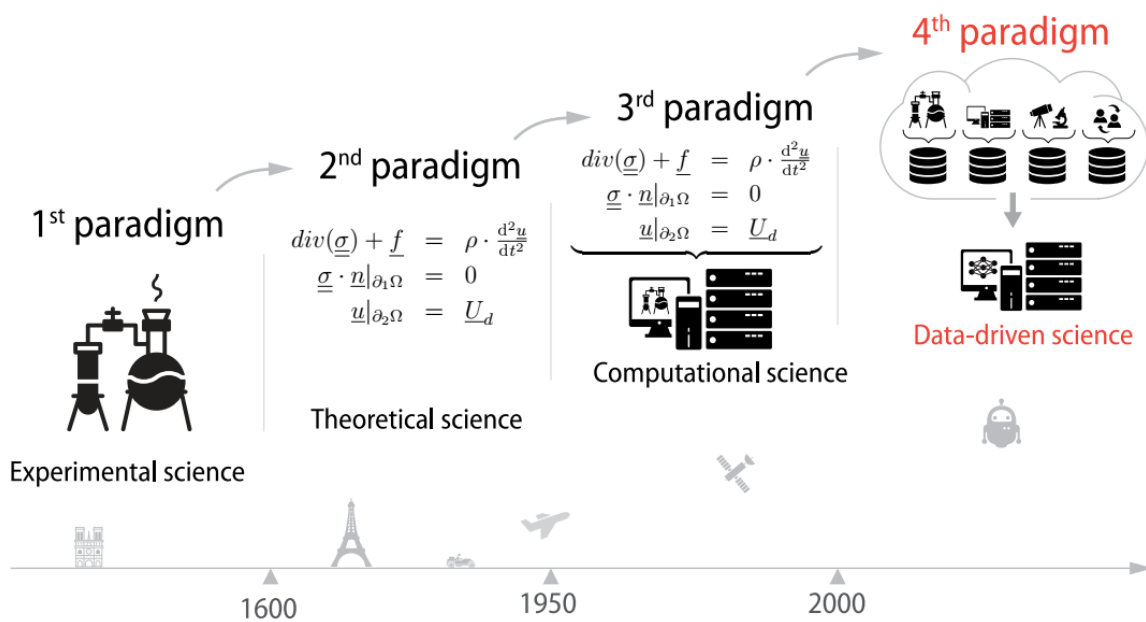


Figure 1 – Appréhension des phénomènes scientifiques

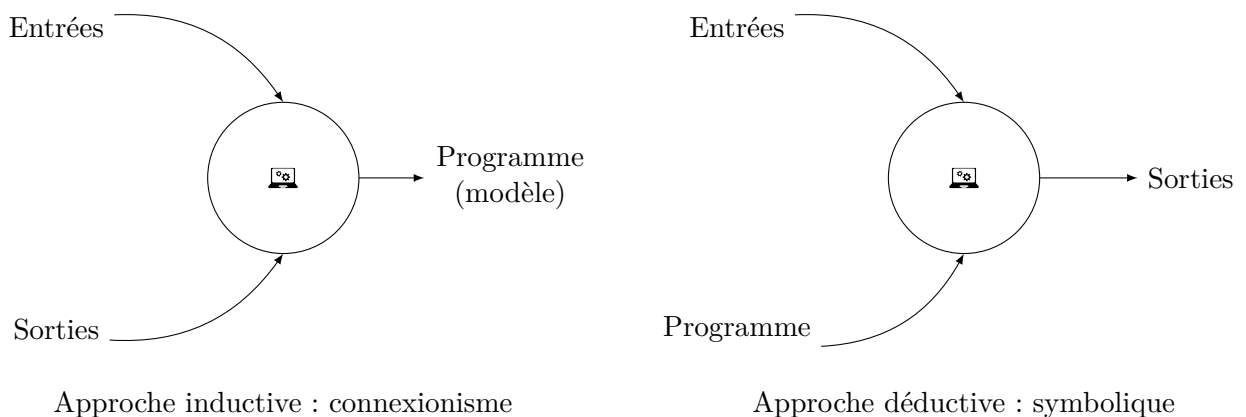
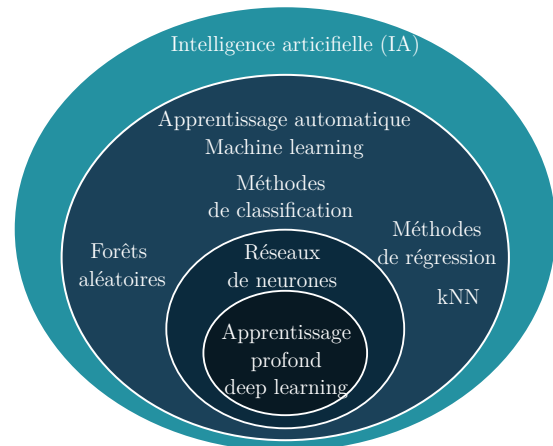


Figure 2 – Schéma de la différence l'approche connexioniste et l'approche symbolique

2 Vocabulaire de différentes approches en intelligence artificielle

2.1 Intelligence artificielle avec le machine learning

l'intelligence artificielle a pour but de simuler un ou des comportements humains. À l'heure actuelle, il est encore difficile de développer des intelligences artificielles dites fortes. Dans le domaine de l'intelligence artificielle, des algorithmes sont développés et notamment le machine learning regroupant notamment les méthodes à réseaux de neurones dans lesquelles le deep-learning est développé. Le deep-learning est notamment utilisé pour la reconnaissance d'images.



2.2 Classification des algorithmes de machine learning

Dans le programme de CPGE, on s'intéresse à quelques algorithmes de machine learning :

- Méthode des k plus proches voisins (Info Commun et SII) ;
- Méthode des k -moyennes (Info Commun) ;
- Réseaux de neurones (SII) ;

Cependant, ces trois algorithmes ne sont qu'une partie de l'ensemble des algorithmes possibles pour réaliser du machine learning.

Il existe néanmoins trois grandes catégories d'algorithmes de machine learning qui ont principalement deux buts : la classification ou la régression.

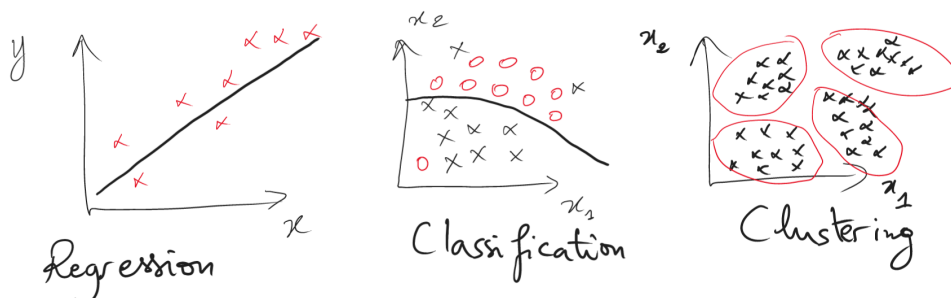


Figure 3 – Trois grandes types de problèmes : classification , Régression ou Clustering

- **Apprentissage supervisé** : 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. 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.

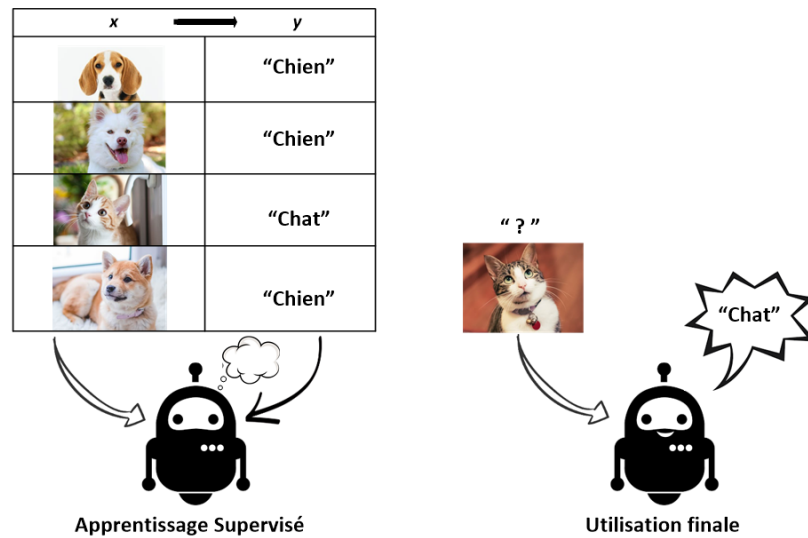


Figure 4 – Principe de l'apprentissage supervisé

- Apprentissage non supervisé** : 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.

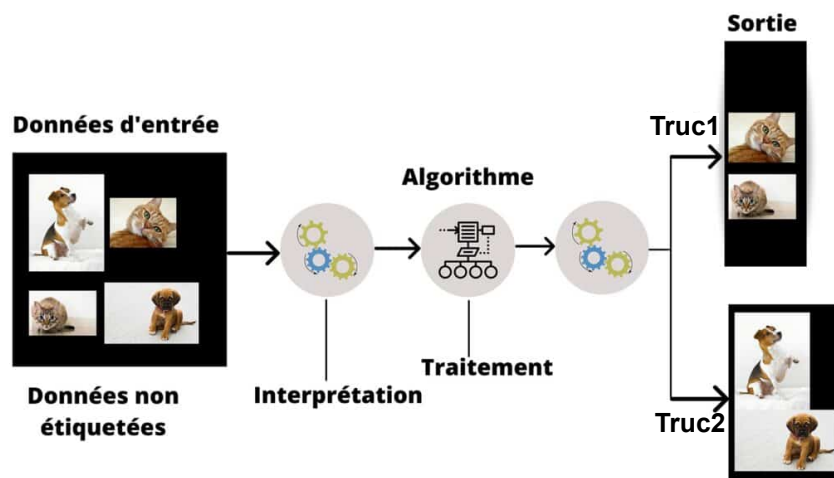


Figure 5 – Principe de l'apprentissage non supervisé

- Apprentissage par renforcement** : 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.

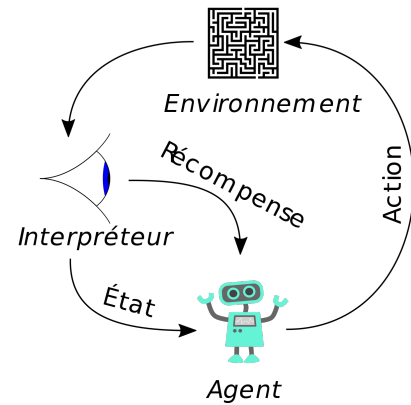


Figure 6 – Principe de l'apprentissage par renforcement

Voici un schéma synoptique des différents algorithmes de machine learning existants.

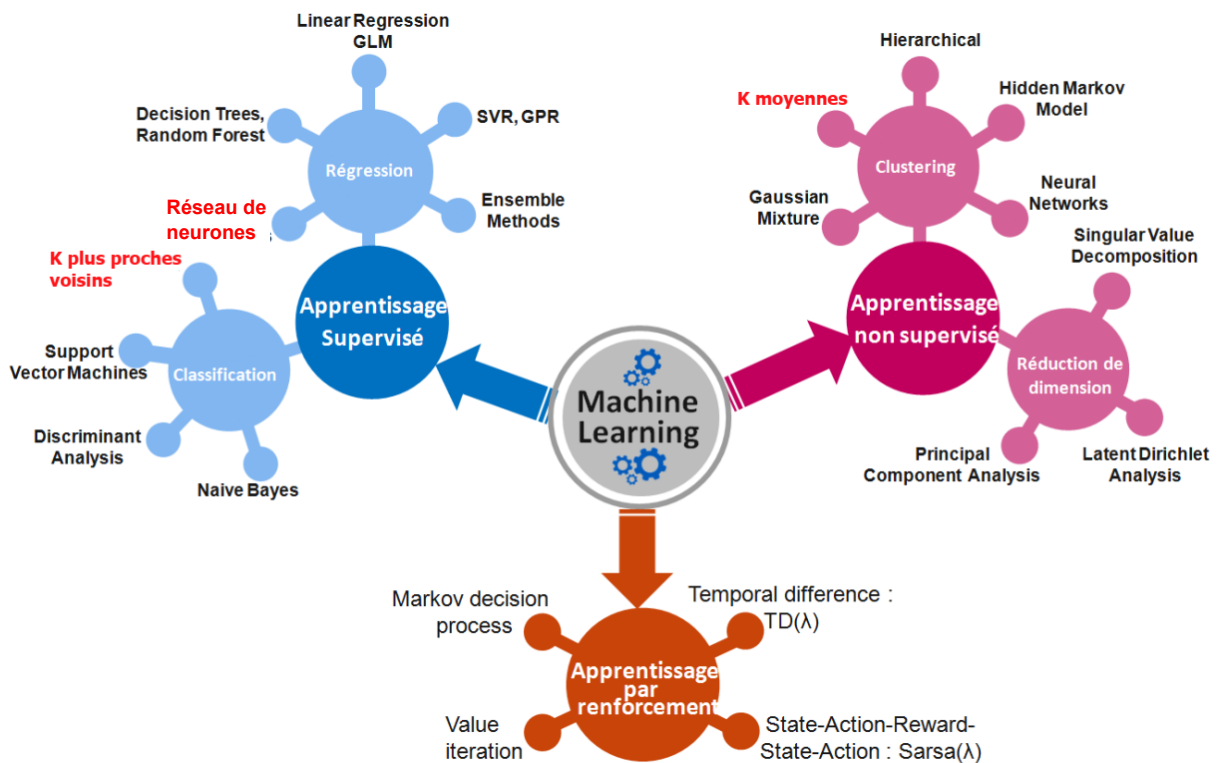


Figure 7 – Les différents algorithmes d'IA

3 k plus proches voisins : kNN

3.1 Principe de l'algorithme des k plus proches voisins

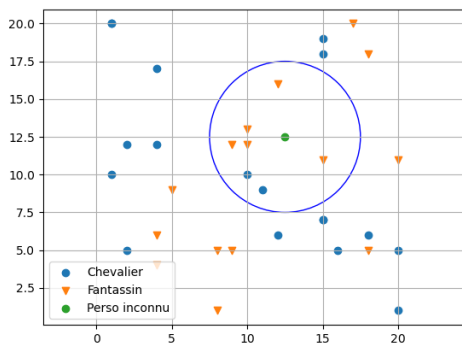
3.1.1 Illustration de la méthode des k plus proches voisins

L'algorithme des k plus proches voisins (*k-nearest neighbors* : kNN en anglais) est un algorithme qui peut être utilisé pour de la **classification** ou de la **régression**. Il est étiqueté comme étant un algorithme à apprentissage **supervisé**. C'est effectivement un algorithme utilisant des données labellisées/étiquetées/classifiées (il est donc bien supervisé) mais il n'y a en rien une phase d'apprentissage.

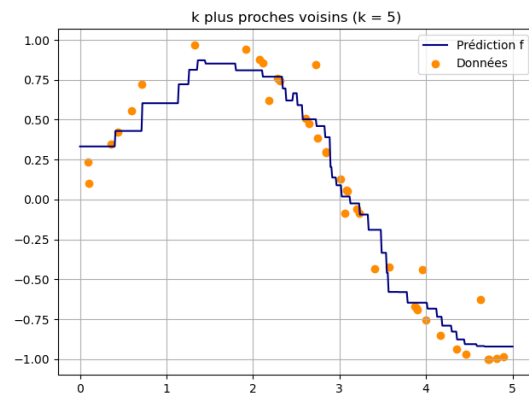
C'est certainement l'un des algorithmes d'intelligence artificielle le plus simple à appréhender.

L'algorithme des k plus proches voisins est un type spécial d'algorithme qui n'utilise pas de modèle statistique. Il est "**non paramétrique**" et il se base uniquement sur les données d'entraînement. Ce type d'algorithme est appelé *memory-based*.

Classification



Régression



3.1.2 Algorithmes

On présente sous forme de pseudo-code les deux algorithmes des k plus proches voisins dans le cas de la classification et de la régression.

Initialisation;

Données : $\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_p, y_p)\};$

– p observations en n dimensions : $\mathbf{x}_i \in \mathbb{R}^n;$

– p étiquettes : $y_i \in \mathcal{Y}$. (Les étiquettes dans l'exemple précédent étaient **chevalier** et **fantassin** ;

) **Objectif :** Déterminer l'étiquette associée à \mathbf{x}^* ;

pour i variant de 1 à p **faire**

 | Établir le tableau de taille p des distances entre \mathbf{x}^* et \mathbf{x}_i ;

fin

Déterminer les k points \mathbf{x}_i les plus proches de \mathbf{x}^* ;

Par le vote majoritaire parmi les k plus proches voisins, on détermine l'étiquette de \mathbf{x}^* .

Cette étiquette est notée y^* ;

return y^*

Algorithme 1 : Algorithme des k plus proches voisins : Classification

Initialisation;**Données :** $\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_p, y_p)\}$;– p observations en n dimensions : $\mathbf{x}_i \in \mathbb{R}^n$;– p étiquettes : $f(\mathbf{x}_i) = y_i \in \mathcal{Y}$;**Objectif :** Estimer la valeur de $f(\mathbf{x}^*)$;**pour** i variant de 1 à p **faire**| Établir le tableau de taille p , des distances entre \mathbf{x}^* et \mathbf{x}_i ;**fin**Déterminer les k points \mathbf{x}_{ki} les plus proches de \mathbf{x}^* avec \mathbf{x}_{ki} le $i^{\text{ème}}$ point le plus proche de \mathbf{x}^* parmi les k plus proches ;Estimer la moyenne $f(\mathbf{x}^*) = \frac{1}{k} \sum_{i=1}^k f(\mathbf{x}_{ki})$;**return** $f(\mathbf{x}^*)$ **Algorithme 2 :** Algorithme des k plus proches voisins : Régression**3.1.3 Un exemple**

L'algorithme des k plus proches voisins est ici présenté pour la reconnaissance de caractères. On utilise pour cela la base de données `mnist` disponible sous Python.

```

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

```

Échantillonnages des données

Cette base de données est composée de 70 000 images de taille 28×28 pixels. On utilisera ici que 15% des images, soit 10 500 (on utilise ici que 15% pour des raisons de temps de calcul). Pour réaliser le code des k plus proches voisins, on utilise les fonctions de la bibliothèque `sklearn`.

```

1 | # Importation de la bibliotheque mnist. Cela peut prendre du temps.
2 | from sklearn.datasets import fetch_openml
3 | mnist = fetch_openml('mnist_784', version=1)
4 | # Importation d'une fonction permettant de separer les donnees en deux groupes.
5 | from sklearn.model_selection import train_test_split
6 | vect_x_utile,vect_x_non_utile,vect_y_utile,vect_y_non_utile=train_test_split
   | (mnist.data, mnist.target, train_size=0.15) # 15% des donnees seront utilisees et
   | stockees dans vect_x_utile (28x28 pixels) et vect_y_utile le label de l'image
1 | vect_x_train, vect_x_test, vect_y_train, vect_y_test = train_test_split(vect_x
   | _utile, vect_y_utile, train_size=0.8)

```

Algorithme des k plus proches voisins, pour un nombre k fixé

```

1 | from sklearn import neighbors # Importation de la bibliotheque utile pour l'
   | algorithme des k plus proches voisins
2 | knn = neighbors.KNeighborsClassifier(3) # Creation du modele (ici on fixe k=3)
3 | knn.fit(vect_x_train,vect_y_train) # Sauvegarde des donnees utiles

```

```
4 predictions=knn.predict(vect_x_test) # Predictions des vect_y_test pour les donnees
    vect_x_test
5 liste_prediction_correkte=[predictions[i]==vect_y_test.values[i] for i in range(len(
    predictions))] # liste
6 pourcentage_erreur=(1-sum(liste_prediction_correkte)/len(liste_prediction_correkte)
    )*100
7 # Pourcentage d'erreur dans la prediction
8 print("pourcentage d'erreur : ", pourcentage_erreur)
```

Le résultat obtenu est un pourcentage d'erreur de l'ordre de 6%.

3.2 Algorithme des k plus proches voisins, recherche de la meilleure valeur de k

```
1 # Les données vect_x_train, etiquettes_train, vect_x_test et etiquettes_test sont
    disponibles.
2 import matplotlib.pyplot as plt
3 errors = []
4 for k in range(2,15):
5 knn = neighbors.KNeighborsClassifier(k) # Création du modèle
6 knn.fit(vect_x_train, vect_y_train) # Sauvegarde des données utiles
7 errors.append(100*(1 - knn.score(vect_x_test, vect_y_test))) # Calculs de l'erreur
    pour chaque valeur de k (remplacez aussi par score)
8 plt.plot(range(2,15), errors, 'o-')
9 plt.xlabel("Nombre k")
10 plt.ylabel("Erreur en %")
11 plt.show()
```

On obtient alors :

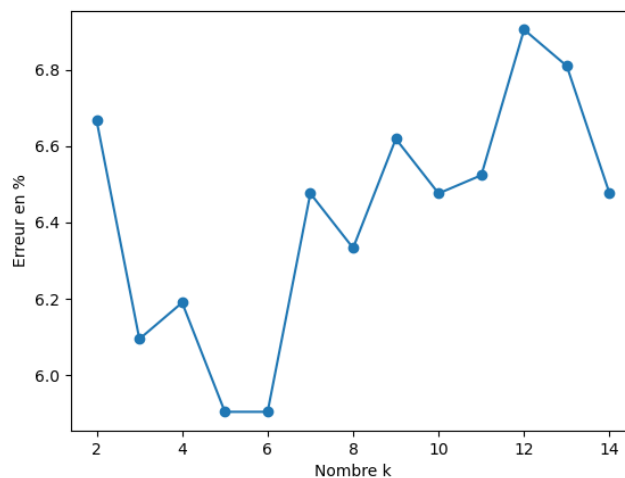


Figure 8 – Erreur de prédiction en fonction de k

3.3 Matrice de confusion

Grâce à la fonction `score`, on peut connaître la qualité de prédiction du classifieur. Cependant, il pourrait être très intéressant de savoir où les erreurs sont produites et comment. Pour cela, on

va utiliser la **matrice de confusion**.

```

1 from sklearn.metrics import confusion_matrix, plot_confusion_matrix # pour utiliser
  la matrice et la tracer
2 import matplotlib.pyplot as plt
3 # Les données vect_x_train, vect_y_train, vect_x_test et vect_y_test sont
  disponibles.
4 knn = neighbors.KNeighborsClassifier(3) # Création du modèle
5 knn.fit(vect_x_train, vect_y_train) #Entraînement
6 class_names=list(range(10)) # liste de nombre de 0 à 9
7 disp = plot_confusion_matrix(knn, vect_x_test, vect_y_test, display_labels= class_
  names, cmap=plt.cm.Blues, normalize=None)
8 disp.ax_.set_title("Matrice de confusion, sans normalisation")
9 print(disp.confusion_matrix)
10 disp = plot_confusion_matrix(knn, vect_x_test, vect_y_test, display_labels= class_
  names, cmap=plt.cm.Blues, normalize='true')
11 disp.ax_.set_title("Matrice de confusion normalis\ee")
12 print(disp.confusion_matrix)

```

Dans la fonction `plot_confusion_matrix`, l'option `cmap` permet de colorer la matrice de confusion, et on peut normaliser les données ou non.

Les résultats obtenus sont les suivants :

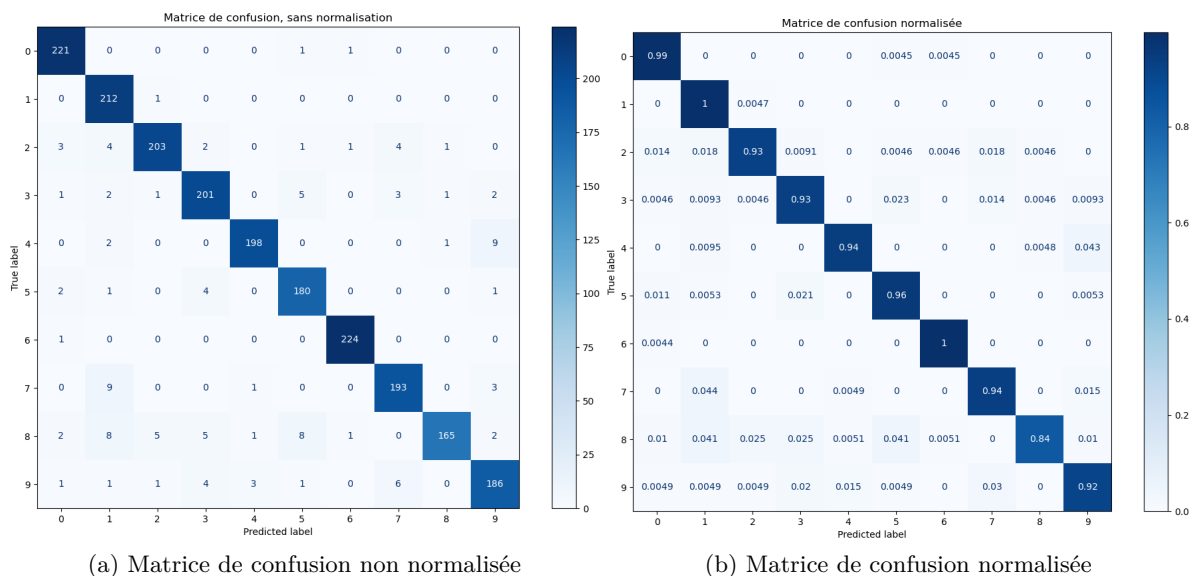


Figure 9 – Les matrices de confusions

3.4 Sensibilité et spécificité

Il en résulte la matrice de confusion ci-dessous :

- La **sensibilité d'un critère** est la probabilité que celui-ci détecte la catégorie *positive* parmi les exemples qui le sont réellement. Il s'agit donc du rapport $\frac{VP}{VP + FN}$.

		nature du chiffre prédit	
		correcte	incorrecte
nature réelle du chiffre	correcte	VP	FN
	incorrecte	FP	VN

- La **spécificité d'un critère** est la probabilité que celui-ci détecte la catégorie *négative* parmi les exemples qui le sont réellement. Il s'agit donc du rapport $\frac{VN}{FP + VN}$.

Avec ces notations, la **justesse (Accuracy)** vaut

$$Accuracy = \frac{VP + VN}{VP + VN + FP + FN}$$

et porte donc une information différente.

Pour le chiffre 8, on obtient la matrice suivante :

		nature du chiffre 8 prédit	
		correcte	incorrecte
nature réelle du chiffre 8	correcte	165	32
	incorrecte	3	1900

3.5 Importance de la mise en forme des données

De par les échelles de certaines données, celles-ci peuvent avoir un poids très important qui n'est pas forcément justifié. Par exemple, dans le domaine médical si on s'intéresse à la fréquence cardiaque (autour de 70 battements/min) et au taux de cholestérol (variant autour de 1,5 à 2,5 g/L). On comprend qu'une variation d'une unité sur le battement cardiaque aura une faible influence sur l'état de santé du patient. Par contre, une variation d'une unité sur le taux de cholestérol a une influence significative.

Voici les données de quatre patients (ceci est bien sûr un exemple fictif) :

Numéro	Fréquence cardiaque (bat/min)	Taux de cholestérol (g/L)	État santé
1	80	1,8	Bon
2	70	2,3	Mauvais
3	65	1,6	Bon
4	90	2,1	Mauvais

Pour réaliser un algorithme des k plus proches voisins de bonne qualité, il est nécessaire que les données soient dans des unités ou à une échelle comparable. Pour cela, on retiendra principalement deux possibilités :

- Ramener les données à une échelle commune (généralement entre 0 et 1)
- Normer les données par rapport à leur écart-type (représentant la dispersion moyenne de celles-ci)

4 Régression

4.1 Intérêt de déterminer un modèle de données

4.2 Présentation

Lors d'une étude expérimentale, des points de mesures sont réalisés (en figure 10a). Afin de traiter ces données, celles-ci sont alors souvent représentées graphiquement afin de mieux les visualiser.

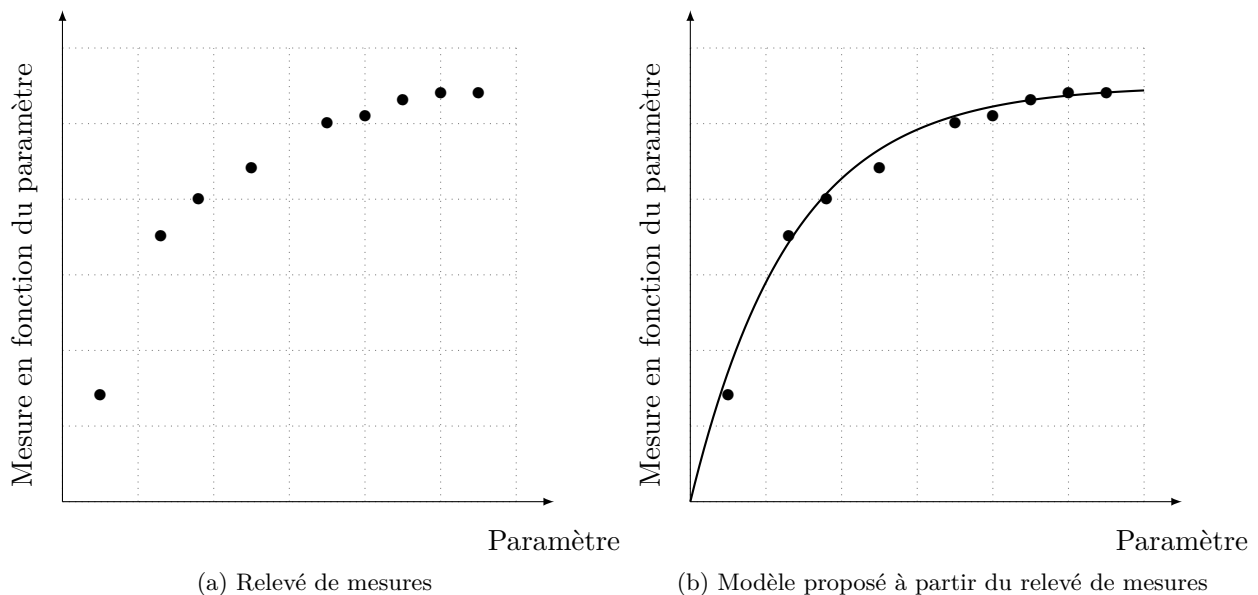


Figure 10 – Exemple (fictif) de relevé de mesures

L'ingénieur (ou plus largement le scientifique) peut alors à partir de ces données tenter de définir un modèle de comportement de ce qui a été observé (figure 10b). C'est-à-dire en déduire une équation liant les paramètres expérimentaux et les mesures obtenues.

4.2.1 Différents modèles possibles

La méthode permettant de déterminer une équation à partir d'un relevé de points n'est cependant pas "magique" et nécessite en fait de faire un choix sur le type de modèle que l'on va utiliser pour déterminer la "meilleure" équation représentative des données disponibles.

Exemple

En reprenant les données de l'exemple de la figure 10b, on propose deux modèles d'interpolation/régression de données. On ne notera ces deux modèles $\tilde{f}_1(x)$ et $\tilde{f}_2(x)$

- un modèle régressif de type exponentielle : $\tilde{f}_1(x) = \gamma_1 \times (1 - e^{\gamma_2 \cdot x})$
- un modèle interpolant de type polynomiale : $\tilde{f}_2(x) = \sum_{i=0}^8 \beta_i \times x^i$

Après optimisation des paramètres γ_i et β_i , le résultat obtenu est :

4.2.2 Critère d'optimisation des paramètres du modèle choisi

Définition de l'erreur quadratique moyenne On dispose d'un ensemble de données $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_p\}$ avec $\mathbf{x}_i \in \mathbb{R}^n$. \mathbf{x}_i peut donc être considéré comme un vecteur de dimension n .

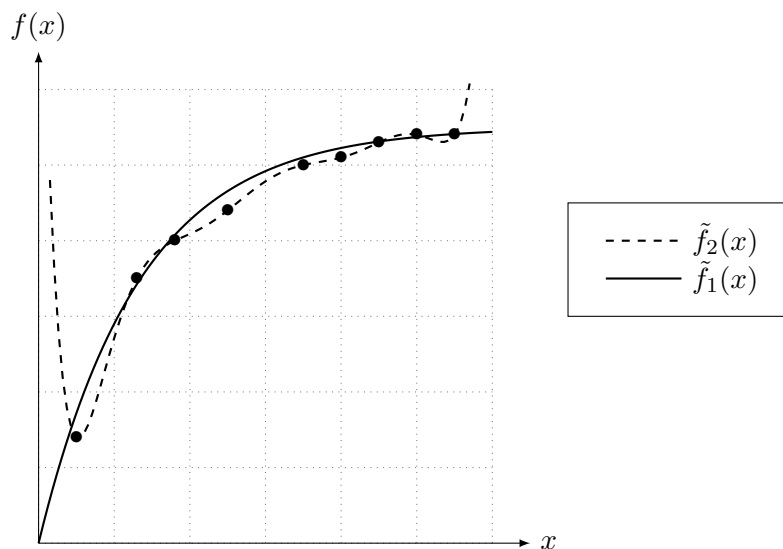


Figure 11 – Deux modèles interpolant ou régressif de données

On vérifie alors :

$$\mathbf{x}_i = [\mathbf{x}_{i_1}, \mathbf{x}_{i_2}, \dots, \mathbf{x}_{i_n}]$$

Pour les données de l'ensemble \mathcal{X} , on dispose de leurs évaluations qui sont stockées dans l'ensemble $\mathcal{Y} = \{y_1, y_2, \dots, y_p\}$. On vérifie $y_i \in \mathbb{R}$. (On peut généraliser à $\mathbf{y}_i \in \mathbb{R}^k$).

Dans ce cours, quel que soit le modèle utilisé pour établir une régression des données, on s'appuie sur **la minimisation de l'erreur quadratique moyenne**. On dispose d'un modèle \tilde{f} dont on cherche à optimiser les paramètres **param**. (**param** est un vecteur des paramètres que l'on cherche à optimiser). On pose alors :

$$\tilde{y}_i = \tilde{f}(\mathbf{x}_i, \text{param})$$

L'erreur quadratique moyenne (*mean square error* en anglais) est donnée par :

$$\begin{aligned} MSE(\text{param}) &= \frac{1}{p} \sum_{i=1}^p \| y_i - \tilde{y}_i \|^2 \\ &= \frac{1}{p} \sum_{i=1}^p \| y_i - \tilde{f}(\mathbf{x}_i, \text{param}) \|^2 \end{aligned}$$

Résolution du problème

Afin de trouver le vecteur **param** le plus adapté aux données \mathcal{X} et leurs évaluation \mathcal{Y} , il faut résoudre le problème suivant :

$$\arg \min_{\text{param}} \left(\frac{1}{p} \sum_{i=1}^p \| y_i - \tilde{f}(\mathbf{x}_i, \text{param}) \|^2 \right)$$

Résoudre ce problème revient à chercher la solution d'un problème de minimisation. On peut définir différents types de minimum pour une fonction f quelconque . (voir figure 12)

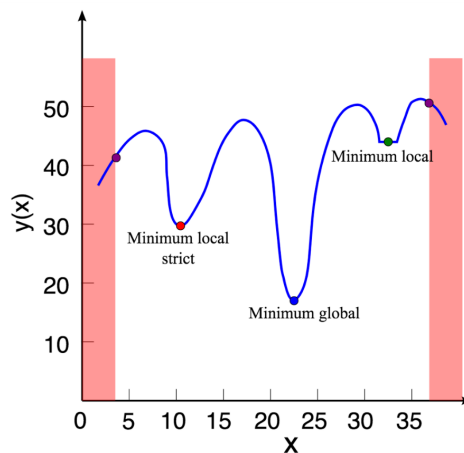


Figure 12 – Illustration de différents types de minima

Pour résoudre, ce problème de minimisation, on peut utiliser différents algorithmes. L'un des plus simples à utiliser est l'algorithme de descente de gradient à pas constant.

Initialisation;

Données : Une fonction f_{onc} dont on cherche un minimum. Un point $\mathbf{x}_{init} \in \mathbb{R}^n$ avec lequel on initialise l'algorithme. α est un paramètre appelé le pas. $\alpha > 0$. Un paramètre $\varepsilon > 0$ servant de critère d'arrêt;

$\mathbf{x}_{sol} \leftarrow \mathbf{x}_{init}$;

tant que $\|\nabla f_{onc}(\mathbf{x}_{sol})\| \geq \varepsilon$ **faire**

$\mathbf{x}_{sol} \leftarrow \mathbf{x}_{sol} - \alpha \nabla f_{onc}(\mathbf{x}_{sol})$;

fin

return x_{sol}

Algorithme 3 : Algorithme de descente de gradient à pas fixe

4.2.2.1 Qualité du modèle : Calcul du coefficient de détermination R^2

Afin d'avoir une estimation de la qualité de notre régression, on utilise traditionnellement le coefficient de détermination R^2 . Ce coefficient se calcule de la façon suivante :

$$R^2 = 1 - \frac{\sum_{i=1}^p (y_i - \tilde{f}(x_i))^2}{\sum_{i=1}^p (y_i - \bar{y})^2} \quad \text{avec } \bar{y} \text{ la moyenne des données } y_i$$

Il représente le degré d'ajustement du modèle par rapport aux données utilisées. C'est donc un bon indicateur de la pertinence de votre modèle, **mais** ce n'est qu'un indicateur.

4.3 La régression linéaire simple

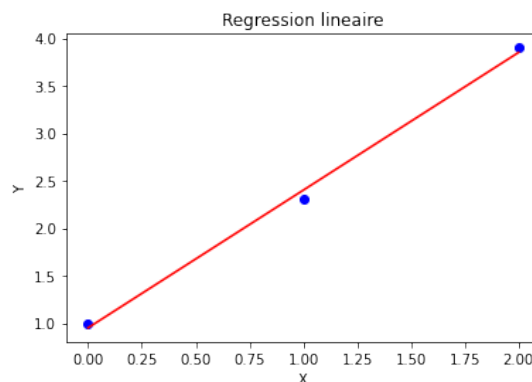
- Cas de la régression monovariante : $\tilde{f}(x) = a.x$
- Cas de la régression multivariante : $\tilde{f}(x) = a.x + b$

4.3.1 Exemple de la régression multivariante : $\tilde{f}(x) = a.x + b$

Soient $\mathcal{X} = \{0, 1, 2\}$ et $\mathcal{Y} = \{1, 2.3, 3.9\}$ avec \mathcal{X} les paramètres évalués et \mathcal{Y} leurs évaluations. On souhaite établir un modèle de régression du type $\tilde{f}(x) = a.x$ à partir de ces données.

```
1 from sklearn.linear_model import LinearRegression
2 import numpy as np # pour les listes de donnees
3 import matplotlib.pyplot as plt # pour les courbes
4 X = np.array([0, 1, 2]).reshape((-1, 1)) # Donnees d'entree normalisees
5 Y = np.array([1, 2.3, 3.9]) #Sortie
6 model = LinearRegression().fit(X, Y) # Entraînement
7 r_sq = model.score(X, Y)
8 print("R^2:", r_sq)
9 print("constante:", model.intercept_)
10 print("pente:", model.coef_)
11 X_new = np.array([3]).reshape((-1, 1))
12 Y_new = model.predict(X_new)
13 print(Y_new, 'pour x = ', X_new) # prediction de nouvelle valeur
14 plt.scatter(X, Y, color="blue")
15 Y_pred = model.predict(X.reshape((-1, 1)))
16 plt.plot(X, Y_pred, color="red")
17 plt.xlabel("X")
18 plt.ylabel("Y")
19 plt.title("Regression lineaire ")
20 plt.show()
```

```
R2: 0.9964454976303317
constante: 0.95000000000000006
pente: [1.45]
[5.3] pour x = [[3]]
```



4.4 Régression linéaire multiple

4.4.1 Méthode de résolution directe

Nous venons de traiter des problèmes de régression pour modéliser un comportement/un problème/une fonction ne dépendant que d'une variable (la variable x). Les paramètres a et b déterminés précédemment ne sont que les paramètres du modèle de régression déterminé.

Nous allons dans cette subsection, nous intéresser au problème dépendant de plusieurs variables.

Le problème est le suivant : On dispose d'un ensemble de données $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_p\}$ avec $\mathbf{x}_i \in \mathbb{R}^n$. \mathbf{x}_i peut donc être considéré comme un vecteur de dimension n . On vérifie alors :

$$\mathbf{x}_i = [x_{i1}, x_{i2}, \dots, x_{in}]$$

Pour cet ensemble de données \mathcal{X} , on dispose de leurs évaluations qui sont stockées dans l'ensemble $\mathcal{Y} = \{y_1, y_2, \dots, y_p\}$. On vérifie $y_i \in \mathbb{R}$.

Dans le cas de régression linéaire multiple, on cherche donc à établir un modèle

$$\tilde{f}(\mathbf{x}, \mathbf{B}) = \beta_0 + \beta_1 \cdot x_1 + \beta_2 \cdot x_2 + \cdots + \beta_n \cdot x_n \text{ avec } \mathbf{x} \in \mathbb{R}^n \text{ et } \mathbf{B} = [\beta_0, \beta_1, \cdots, \beta_n]^T \in \mathbb{R}^{n+1}$$

L'erreur quadratique moyenne est donc définie par :

$$MSE(\mathbf{B}) = \frac{1}{p} \sum_{i=1}^p \| y_i - \tilde{f}(\mathbf{x}_i, \mathbf{B}) \|^2$$

De nouveau, on peut représenter ce problème de manière matricielle.

$$\text{On pose } A = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1n} \\ 1 & x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 1 & x_{p1} & x_{p2} & \cdots & x_{pn} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} \text{ et } \mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{bmatrix}$$

De ce fait :

$$\begin{aligned} MSE(\mathbf{B}) &= \frac{1}{p} \sum_{i=1}^p \| y_i - \tilde{f}(\mathbf{x}_i, \mathbf{B}) \|^2 \\ &= \frac{1}{p} \| \mathbf{A} \cdot \mathbf{B} - \mathbf{Y} \|^2 \end{aligned}$$

On cherche à résoudre le problème :

$$\arg \min_{\mathbf{B}} \| \mathbf{A} \cdot \mathbf{B} - \mathbf{Y} \|^2 \iff \arg \min_{\mathbf{B}} (\mathbf{A} \cdot \mathbf{B} - \mathbf{Y})^T (\mathbf{A} \cdot \mathbf{B} - \mathbf{Y})$$

La solution de ce problème est :

$$\mathbf{B} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{Y}$$

Conclusion

Lorsque le problème de régression que l'on cherche à résoudre est constitué de p données en nombre important et que la dimension du problème n est aussi importante, les différentes opérations matricielles prennent un temps non négligeable. C'est pour cela qu'une approche itérative (comme celle de l'algorithme à descente de gradient à pas fixe) est une solution. Il y a plusieurs variantes sur ce genre d'algorithme (pas variable, pas adapté, etc...). Une autre méthode efficace sont les méthodes à région de confiance.

5 Réseau de Neurones : Neural Network

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

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

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

Voilà à quoi ressemble un Réseau de Neurones :

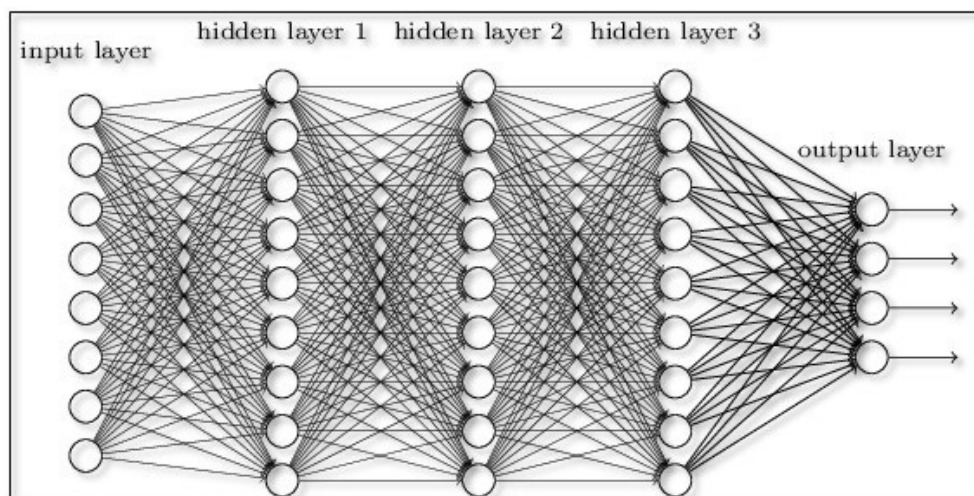


Figure 13 – Principaux composants d'un réseau de neurones

5.1 Réseau de Neurone à 1 Neurone : Le perceptron

Le réseau de Neurones le plus simple qui existe porte le nom de perceptron. Les entrées du neurone sont les features x multipliées par des paramètres w à apprendre. Le calcul effectué par le neurone peut être divisé en deux étapes :

1. Le neurone calcule la somme z de toutes les entrées $z = \sum_1^n (x_i \cdot w_i) + b$. C'est un calcul linéaire
2. Le neurone passe z dans sa fonction d'activation. Ici la fonction heaviside (fonction unité). C'est un calcul non-linéaire.

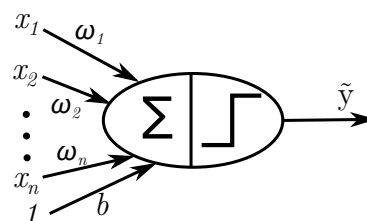

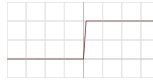
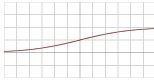
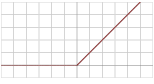


Figure 14 – Etapes d'activation d'un perceptron

Voici quelques fonctions d'activation classiquement utilisés :

Identité	Heaviside	Logistique (sigmoïde)	Unité de rectification linéaire (ReLU)
			
$f(x) = x$	$f(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$	$f(x) = \frac{1}{1 + e^{-x}}$	$f(x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases}$

5.2 Perceptron multi-couches

Étudions l'exemple qui peut paraître très simple du OU EXCLUSIF. Les données d'entrées sont deux bits x_1 et x_2 . La sortie y vaut 0 ou 1, réponse de l'opérateur booléen XOR aux bits x_1 et x_2 .

$$y = x_1 \oplus x_2$$

Les données et leur évaluation sont les suivantes :

\mathcal{X}	\mathcal{Y}
[0,0]	0 (●)
[0,1]	1 (■)
[1,0]	1 (■)
[1,1]	0 (●)

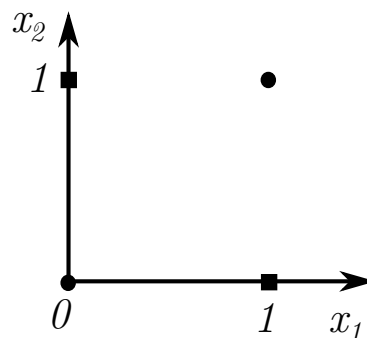


Figure 15 – Représentation graphique de XOR

L'exemple précédent, montre que si l'on est capable de représenter par un classifieur linéaire les opérateurs AND et OR, alors on peut créer $AND(x_1, x_2)$ et $OR(x_1, x_2)$. Il suffit ensuite de rechercher le classifieur linéaire adapté à ce nouveau problème.

L'idée (ou l'intuition) qui se cache derrière un réseau de neurones (ou un perceptron multi-couches ici) est de transformer l'espace initial des données afin de rendre celles-ci finalement séparables.

Dans le cadre du problème XOR, voici un réseau de neurones permettant de le résoudre.

Ce réseau de neurones est composé de 3 couches :

- Une couche d'entrée, qui correspond tout simplement aux données qui vont être traitées ;
- Une couche cachée dont on ne connaît pas les prédictions en sortie des neurones

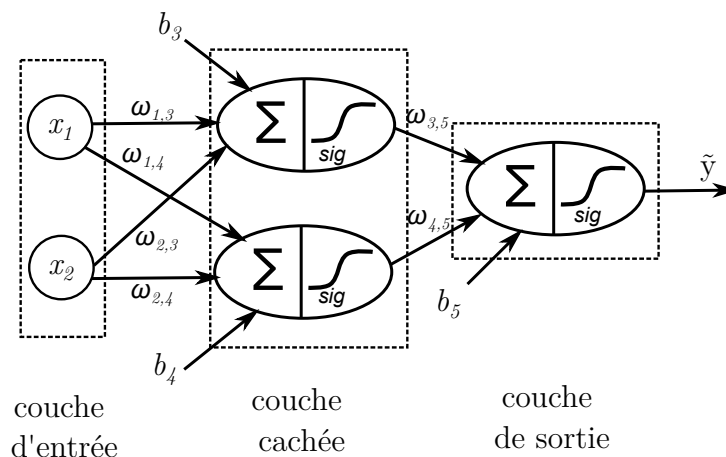


Figure 16 – Résolution du problème XOR par un perceptron multi-couches

- Une couche de sortie dont on connaît les évaluations y associées aux données x_1 et x_2 . Lors de l'entraînement du réseau de neurones, l'objectif recherché sera donc que les prédictions \tilde{y} soient les proches possibles de y . Lors d'une prédiction d'un jeu de donnée (x_1, x_2) , si $\tilde{y} < 0.5$ on considère alors que la donnée est catégorisée dans la catégorie 0 (●) et si $\tilde{y} \geq 0.5$ celle-ci est catégorisée dans la catégorie 1 (■).

Remarque

Quand on décrit un réseau de neurones, le nombre de couches de celui-ci est parfois écrit sans tenir compte des couches d'entrée et de sortie. Par exemple, dans notre cas on dira ici qu'il s'agit d'un réseau de neurones à une couche cachée constituée de deux neurones dont la fonction d'activation est la fonction sigmoïde.

Remarque

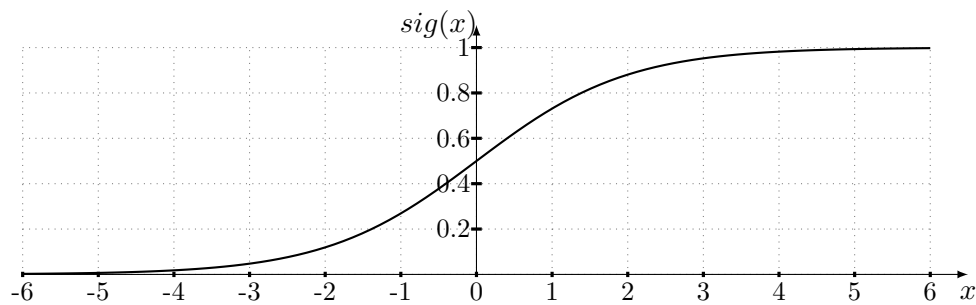
Avec les poids : ● $\omega_{1,3} = 20$, ● $\omega_{2,3} = 20$, ● $\omega_{1,4} = -20$, ● $\omega_{2,4} = -20$, ● $\omega_{3,5} = 20$, ● $\omega_{4,5} = 20$, ● $b_3 = -10$, ● $b_4 = 30$, ● $b_5 = -30$ La décomposition obtenue n'est pas celle décrite précédemment avec les opérateurs OR et AND, mais cela vous a aidé à comprendre l'idée...

5.3 Propagation avant

Reprenons l'exemple précédent avec la valeur des poids $\omega_{i,j}$ et b_k données précédemment.

La fonction *sigmoïde* est définie par :

$$\begin{aligned} sig : \mathbb{R} &\longrightarrow \mathbb{R} \\ x &\longmapsto \frac{1}{1 + e^{-x}} \end{aligned}$$



Effectuer la prédiction \tilde{y} obtenu pour les quatre couples $(x_1, x_2) : \{(0,0), (0,1), (1,0), (1,1)\}$

Pour le couple (0,0) :

- Pour le neurone 3, la valeur en sortie est $sig(-10) =$
- Pour le neurone 4, la valeur en sortie est $sig(30) =$
- Pour le neurone 5, la valeur \tilde{y} en sortie est $sig(\quad) =$

Pour le couple (0,1) :

- Pour le neurone 3, la valeur en sortie est $sig(\quad) =$
- Pour le neurone 4, la valeur en sortie est $sig(\quad) =$
- Pour le neurone 5, la valeur \tilde{y} en sortie est $sig(\quad) =$

Pour le couple (1,0) :

- Pour le neurone 3, la valeur en sortie est $sig(\quad) =$
- Pour le neurone 4, la valeur en sortie est $sig(\quad) =$
- Pour le neurone 5, la valeur \tilde{y} en sortie est $sig(\quad) =$

Pour le couple (1,1) :

- Pour le neurone 3, la valeur en sortie est $sig(\quad) =$
- Pour le neurone 4, la valeur en sortie est $sig(\quad) =$
- Pour le neurone 5, la valeur \tilde{y} en sortie est $sig(\quad) =$

5.4 Phase d'apprentissage et mise à jour des paramètres $\omega_{i,j}$

L'idée de la mise à jour des coefficients $\omega_{i,j}$ (et b_k) se fait selon la règle d'apprentissage du perceptron. Il suffit de la généraliser.

Rappel : La règle d'apprentissage (avec une fonction d'activation *sigmoïde*) est un algorithme de descente de gradient (stochastique) cherchant à miniser une fonction coût. Dans le cadre de la classification (notre cadre d'étude actuel), la fonction que l'on cherche à minimiser est

$$J = Loss = -\log(\text{Maximum Vraisemblance})$$

Afin de s'assurer une cohérence dans les notations, posons le problème précédent sous cette forme afin de n'avoir qu'une notation unique des poids \mathbf{w} (afin de ne pas introduire la notation supplémentaire des biais b_k) :

La règle d'apprentissage est :

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial}{\partial \mathbf{w}} Loss(y_t, \tilde{y}_t) \quad \text{avec } \tilde{y}_t \text{ dépendant de } \mathbf{w}$$

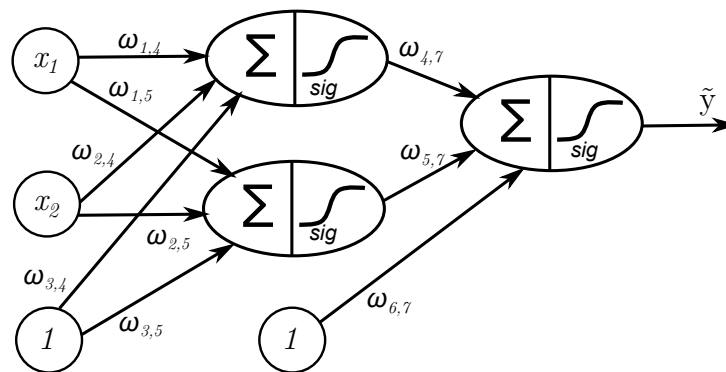


Figure 17 – Réorganisation du réseau

C'est donc bien la même règle que pour le perceptron, mais le calcul s'avère beaucoup plus complexe.

La méthode de **rétropropagation de gradient** est utilisée mais ne sera pas développée ici. Le lecteur est encouragé à s'y intéresser pour sa curiosité mais n'est pas utile pour la suite du propos.

5.4.1 Algorithme de la phase d'apprentissage

Initialisation;

Données : $\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_p, y_p)\}$;

– p observations en n dimensions : $\mathbf{x}_i \in \mathbb{R}^n$;

– p étiquettes : $y_i \in \mathcal{Y}$. ;

Objectif : Trouver les paramètres ω_i les plus adaptés ;

tant que le nombre d'itérations maximal n'est pas atteint **ou** qu'il y a encore des erreurs de prédictions **faire**

pour t variant de 1 à p **faire**

 Phase d'agrégation : calcul de $\mathbf{w} \cdot \mathbf{x}_t$;

 Phase d'activation : calcul de $\tilde{y}_t = h(\mathbf{w} \cdot \mathbf{x}_t)$;

si $\tilde{y}_t \neq y_t$ **alors**

 | $\mathbf{w} \leftarrow \mathbf{w} + \alpha(y_t - \tilde{y}_t) \cdot \mathbf{x}_t$

fin

fin

fin

return \mathbf{w}

Algorithme 4 : Algorithme de la phase d'apprentissage du perceptron

Le paramètre α est appelé **taux d'apprentissage**. La commande $\mathbf{w} \leftarrow \mathbf{w} + \alpha(y_t - \tilde{y}_t) \cdot \mathbf{x}_t$ est appelée **règle d'apprentissage** du Perceptron.

Cette règle d'apprentissage est importante. En effet :

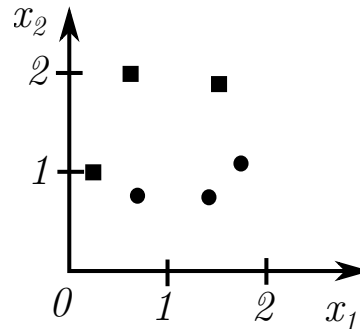
- si $\tilde{y}_t = y_t$, l'ensemble des poids ω_i semble correctement paramétré et il n'est pas nécessaire de les faire évoluer.
- si $\tilde{y}_t \neq y_t$ une évolution des poids est alors à envisager.
 - Dans le cas où $\tilde{y}_t = 1$ et $y_t = 0$. La règle d'apprentissage fait que la valeur des poids ω_i diminue, que le biais b diminue également et que donc le seuil $-b$ augmente. L'activation du neurone est donc rendue plus "difficile".
 - Dans le cas où $\tilde{y}_t = 0$ et $y_t = 1$. La règle d'apprentissage fait que la valeur des poids ω_i augmente, que le biais b augmente également et que donc le seuil $-b$ diminue.

L'activation du neurone est donc rendue plus "facile".

5.5 Exemple de la phase d'apprentissage du perceptron

On considère l'ensemble d'entraînement suivant , avec une fonction d'activation **heaviside** :

\mathcal{X}	\mathcal{Y}
[0.2,1]	1 (■)
[0.5,2]	1 (■)
[0.7,0.8]	0 (●)
[1.4,0.7]	0 (●)
[1.5,1.8]	1 (■)
[1.8,1.1]	0 (●)



On initialise le perceptron de la façon suivante :

ω_1	ω_2	b
1	1	1

L'hyperparamètre du taux d'apprentissage vaut $\alpha = 0.1$

1^{ère} itération : $(\mathbf{x}_1 = [0.2, 1], y_1 = 1)$

- Phase d'agrégation : $\mathbf{w} \cdot \mathbf{x}_1 =$
- Phase d'activation : $\tilde{y}_1 = h(\mathbf{w} \cdot \mathbf{x}_1) =$
- Mise à jour de \mathbf{w} : $\mathbf{w} =$

2^{ème} itération : $(\mathbf{x}_2 = [0.5, 2], y_2 = 1)$

- Phase d'agrégation : $\mathbf{w} \cdot \mathbf{x}_2 =$
- Phase d'activation : $\tilde{y}_2 = h(\mathbf{w} \cdot \mathbf{x}_2) =$
- Mise à jour de \mathbf{w} : $\mathbf{w} =$

3^{ème} itération : $(\mathbf{x}_3 = [0.7, 0.8], y_3 = 0)$

- Phase d'agrégation : $\mathbf{w} \cdot \mathbf{x}_3 =$
- Phase d'activation : $\tilde{y}_3 = h(\mathbf{w} \cdot \mathbf{x}_3) =$

- Mise à jour de \mathbf{w} : $\mathbf{w} =$

4^{ème} itération : $(\mathbf{x}_4 = [1.4, 0.7], y_4 = 0)$

- Phase d'agrégation : $\mathbf{w} \cdot \mathbf{x}_4 =$
- Phase d'activation : $\tilde{y}_4 = h(\mathbf{w} \cdot \mathbf{x}_4) =$
- Mise à jour de \mathbf{w} : $\mathbf{w} =$

5^{ème} itération : $(\mathbf{x}_5 = [1.5, 1.8], y_5 = 1)$

- Phase d'agrégation : $\mathbf{w} \cdot \mathbf{x}_5 =$
- Phase d'activation : $\tilde{y}_5 = h(\mathbf{w} \cdot \mathbf{x}_5) =$
- Mise à jour de \mathbf{w} : $\mathbf{w} =$

6^{ème} itération : $(\mathbf{x}_6 = [1.8, 1.1], y_6 = 0)$

- Phase d'agrégation : $\mathbf{w} \cdot \mathbf{x}_6 =$
- Phase d'activation : $\tilde{y}_6 = h(\mathbf{w} \cdot \mathbf{x}_6) =$
- Mise à jour de \mathbf{w} : $\mathbf{w} =$

5.6 Sous-apprentissage et sur-apprentissage

Définition

On appelle *epoch* un cycle d'apprentissage où tous les poids et tous les biais ont été mis à jour en faisant passer toutes les données du jeu d'entraînement dans les algorithmes de propagation et de rétropropagation.

5.6.1 Présentation du contexte

Afin de bien coder, un algorithme d'intelligence artificielle supervisé, il est nécessaire de l'entraîner : **phase d'apprentissage**. Dans le cas d'un réseau de neurones, un bon apprentissage dépend de plusieurs hyper-paramètres : taux d'apprentissage α , fonctions d'activation utilisées, nombre d'epochs, nombre de neurones cachés, etc. Pour un algorithme des k plus proches voisins, il s'agit du nombre k à choisir.

Afin de valider le choix d'un modèle de réseaux de neurones (valeurs des poids \mathbf{w} , nombres de neurones cachés), on partage en deux le jeu de données initial : un jeu d'entraînement (80% des données), servant à la **phase d'apprentissage** et un jeu test (20% des données) servant à la **phase de validation**.

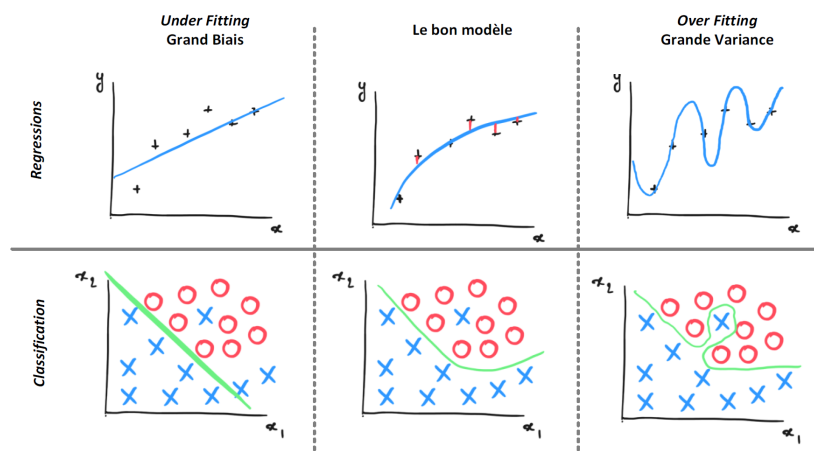
Dans le cadre d'un réseau de neurones, il est important de savoir quand stopper la phase d'apprentissage. Il faut limiter le nombre d'epochs. Faire trop d'epoch sera contre productif (bien que cela ne soit pas forcément évident à première vue), tout comme il est contre productif de mettre trop de couches de réseaux de neurones cachés.

5.6.2 Phases d'apprentissage, de validation et d'inférence

Dans la figure suivante (Figure 5.6.3), on réalise un entraînement, sur le jeu de données (une epoch), c'est la **phase d'apprentissage**. À la fin de l'epoch on dispose d'un premier modèle de réseau de neurones. On détermine alors l'erreur de prédiction commise en utilisant le jeu d'entraînement. **La phase de validation** consiste alors à tester le modèle du réseau de neurones déterminé. On détermine ensuite l'erreur de prédiction commise sur le jeu de test. (On rappelle que le jeu de test ne sert pas à modifier les poids et les biais.) On réalise de même à la fin de l'epoch suivante etc...

Une fois le modèle déterminé, il est alors possible de l'utiliser pour réaliser des prédictions sur des données non connues initialement, c'est **la phase d'inférence**.

5.6.3 Sous-apprentissage et surapprentissage



« Logiquement » l'erreur décroît toujours avec le jeu d'entraînement car la descente du gradient a pour objectif de réduire cette erreur de prédiction.

Sur le jeu de test, l'erreur de prédiction décroît pendant un certain nombre d'epoch puis augmente.

Il existe en fait un stade à partir duquel le réseau de neurones se spécialise **trop** sur le jeu d'entraînement et devient donc incapable de réaliser des prédictions fiables sur un nouveau jeu de données. En effet, le modèle s'est attaché à représenter/comprendre des détails du jeu d'entraînement. On parle de surentraînement ou surapprentissage ou d'overfitting.

La figure ci-contre, permet d'illustrer le phénomène de surapprentissage. La ligne verte représente un modèle sur-appris et la ligne noire représente un modèle régulier. La ligne verte classe trop parfaitement les données d'entraînement, elle généralise mal et donnera de mauvaises prévisions futures avec de nouvelles données. Le modèle vert est donc au final moins bon que le noir.

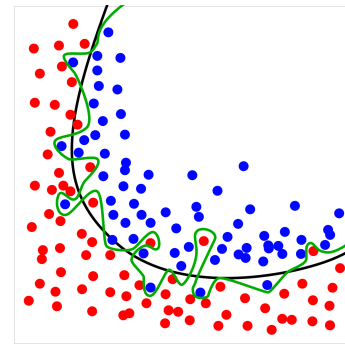


Figure 18 – Illustration du surapprentissage

L'objectif est donc d'arrêter l'entraînement du réseau de neurones lorsque l'erreur de prédiction sur le jeu de test. Il ne faut pas s'arrêter trop tôt car alors le réseau de neurones serait sous-entraîné.

Pour les k -plus proches voisins, le surapprentissage est présent lorsque k est trop petit

Pour les k -plus proches voisins, le sous-apprentissage est présent lorsque k est trop grand

5.7 Réseau de neurone pour des problème de régression de données

Il n'y a pas de grandes différences par rapport à ce que l'on vient de présenter. Seules les fonctions d'activations sont modifiées, on utilisera plutôt des fonctions d'activation linéaires ou ReLU (Rectified Linear Unit). Pour la règle d'apprentissage la fonction coût étudiée n'est pas $-\log(\text{Max Vraisemblance})$ mais tout simplement l'erreur quadratique moyenne.

$$MSE = \frac{1}{p} \sum_{t=1}^p \| y_t - \tilde{y}_t \|^2$$

5.8 Du python :

L'objectif de cet exercice est de réussir à créer un réseau de neurones permettant la reconnaissance de caractères :

```
1 from sklearn.datasets import fetch_openml
2 from sklearn.neural_network import MLPClassifier
3 import numpy as np
4 X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
5 X=np.array(X)
6 y=np.array(y)
7 train_X=X[:60000,:]
8 train_X=train_X.normalise()
9 train_y=y[:60000]
10 test_X=X[60000:70000,]
11 test_y=y[60000:70000]
12 NN=MLPClassifier(activation='relu',hidden_layer_sizes=(50,50,50,),solver='sgd',max
   _iter=50)
13 NN=NN.fit(train_X/255,train_y) #255 pour normaliser les données entre 0 et 1
14 NN.score ( test_X/255, test_y ) # estimer la justesse du reséau à predire
   correctement les données test
```

6 Travaux dirigés - pratiques

6.1 Exercice 1

Considérons le jeu de données suivant :

x_1	x_2	x_3	Classe
1	2	3.5	A
2	4	6	A
3	6	8	B
4	8	10	B
5	10	11	B

1. Calculer la distance euclidienne entre les deux premières observations.
2. Déterminer la classe de la troisième observation en utilisant la méthode des k-nearest neighbors avec $k = 3$ et la distance euclidienne comme mesure de similarité.
3. Si on utilise la même méthode avec $k = 1$, quelle serait la classe attribuée à la troisième observation ?

6.2 Application en médecine - CCINP Info PSI 2019

Objectif : L'objectif du travail proposé est de découvrir quelques techniques d'Intelligence Artificielle en s'appuyant sur un problème médical. À partir d'une base de données comportant des propriétés sur le bassin et le rachis lombaire (figures 19), on cherche à déterminer si un patient peut être considéré comme « normal » ou bien si ces données impliquent un développement anormal de type hernie discale (saillie d'un disque intervertébral) ou spondylolisthésis (glissement du corps vertébral par rapport à la vertèbre sous-jacente).

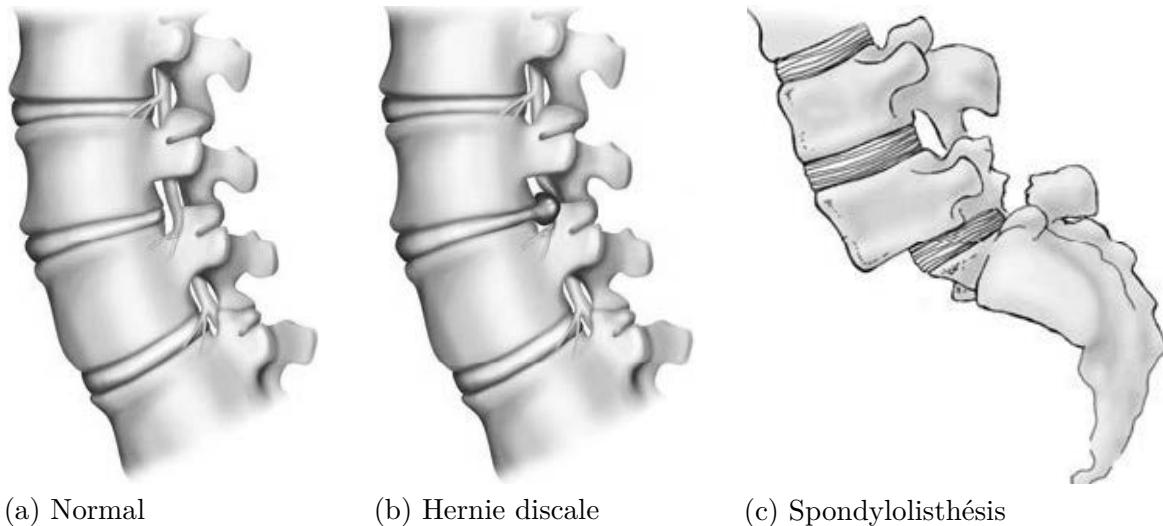


Figure 19 – Différentes configurations des vertèbres

Les 6 attributs considérés dans notre exemple (les n colonnes du tableau) sont définis ci-après :

- angle d'incidence du bassin en $^\circ$;
- angle d'orientation du bassin en $^\circ$;
- angle de lordose lombaire en $^\circ$;
- pente du sacrum en $^\circ$;
- rayon du bassin en mm ;

- distance algébrique de glissement de spondylolisthésis en mm.

Les labels de ces attributs sont stockés dans une liste nommée : `label_attributs = ['incidence_bassin', 'orientation_bassin', 'angle_lordose', 'pente_sacrum', 'rayon_bassin', 'glissement_spon']`.

Le tableau suivant montre les premières valeurs du tableau data.

incidence_bassin	orientation_bassin	angle_lordose	pente_sacrum	rayon_bassin	glissement_spon
63,03	22,55	39,61	40,48	98,67	-0,25
39,06	10,06	25,02	29,0	114,41	4,56
68,83	22,22	50,09	46,61	105,99	-3,53
69,3	24,65	44,31	44,64	101,87	11,21
49,71	9,65	28,32	40,06	108,17	7,92
40,25	13,92	25,12	26,33	130,33	2,23
48,26	16,42	36,33	31,84	94,88	28,34

Table 1 – Données (partielles) des patients à diagnostiquer

Validation de l'algorithme :

Pour tester l'algorithme, on utilise un jeu de données supplémentaires normalisées dont l'état des patients est connu (100 patients). On note `datatest` ces données et `etattest` le vecteur d'état connu pour ces patients. On applique ensuite l'algorithme sur chaque élément de ce jeu de données pour une valeur de K fixée. On définit la fonction suivante qui renvoie une matrice appelée "matrice de confusion".

```
def test_KNN (datatest , etattest , data , etat ,K, nb ) :  
    etatpredit = []  
    for i in range ( len ( datatest ) ) :  
        res = KNN( data , etat , datatest [ i ] ,K, nb )  
        etatpredit . append ( res )  
    mat = np.zeros ( ( nb , nb ) )  
    for i in range ( len ( etattest ) ) :  
        mat [ etattest [ i ] , etatpredit [ i ] ] += 1  
    return mat
```

On obtient pour $K = 8$ la matrice suivante :

$$\begin{matrix} & 23 & 4 & 7 \\ 7 & 11 & 1 & . \\ 5 & 2 & 40 & \end{matrix}$$

Question 1 *Indiquer l'information apportée par la diagonale de la matrice. Exploiter les valeurs de la première ligne de cette matrice en expliquant les informations que l'on peut en tirer. Faire de même avec la première colonne. En déduire à quoi sert cette matrice.*

On peut également tracer l'efficacité de l'algorithme pour différentes valeurs de K sur le jeu de données test (100 éléments sur un échantillon de 310 données). On trace le pourcentage de réussite en fonction de la valeur de K sur la figure 20. 0

Question 2 *Commenter la courbe obtenue et critiquer l'efficacité de l'algorithme.*

Question 3 *Calculer le pourcentage de réussite de la méthode KNN, à partir de la matrice de confusion pour $K = 8$*

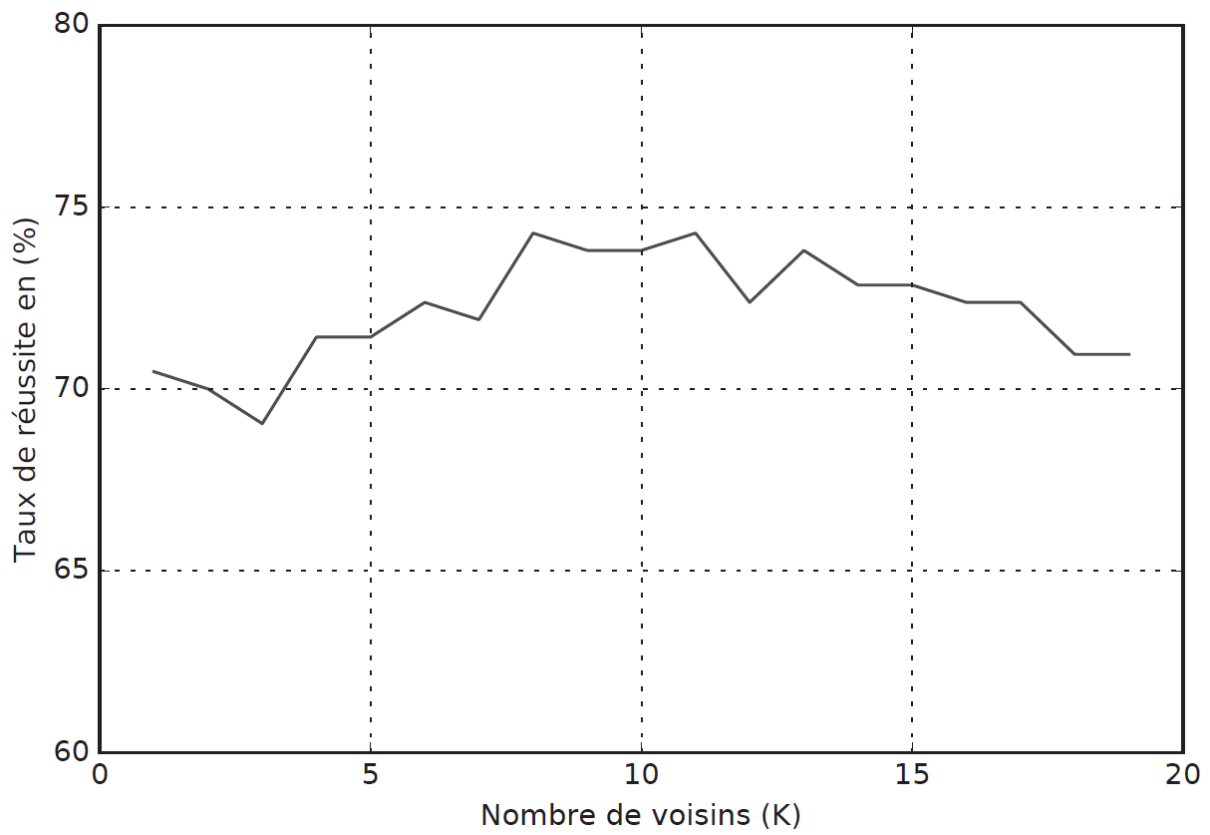


Figure 20 – Pourcentage de réussite de l'algorithme en fonction de K