L'expression intelligence artificielle (IA), a été créée par les américains John McCarthy et Marvin Lee Minsky en 1959. Il s'agit de faire exécuter par des machines des tâches habituellement exécutées par des humains, tâches qui nécessitent des capacités de raisonnement, de mémoire, et d'apprentissage. En particulier on souhaite rendre une machine capable d'accéder à certaines connaissances sans avoir été programmée explicitement pour cela, autrement dit capable d'apprendre par elle-même.

On dit qu'une machine apprend si après certaines expériences elle a augmenté sa capacité à réaliser une tâche. On parle alors d'apprentissage automatique, en anglais *machine learning*.

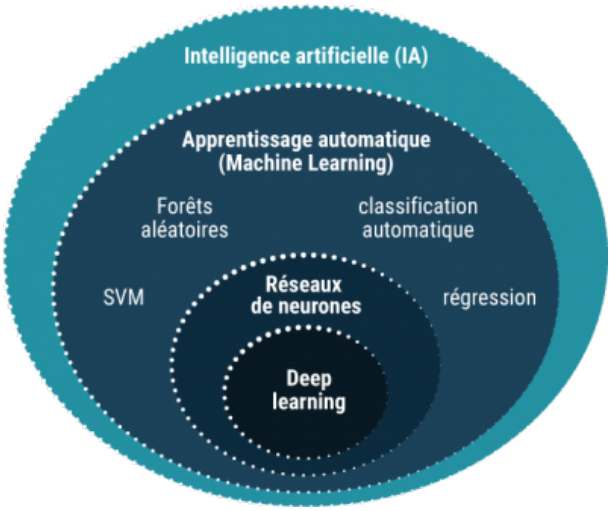

FIGURE 1 – Les différentes entités de l'intelligence artificielle.

L'objectif de ce chapitre est de présenter deux algorithmes d'apprentissage automatique : l'algorithme des k plus proches voisins et l'algorithme des k-moyennes

1. Apprentissage automatique

1.1. Notion d'apprentissage

L'apprentissage automatique, ou *machine learning* en anglais, est un domaine de l'intelligence artificielle. Il s'agit de permettre à une machine de progresser, d'apprendre par elle-même à améliorer son fonctionnement dans un cadre de résolution d'un problème, mais sans jamais la programmer explicitement pour résoudre ce problème. Des outils mathématiques, principalement des outils statistiques, sont appliqués sur des données et les résultats obtenus permettent à la machine d'améliorer peu à peu son efficacité dans la résolution du problème. Par exemple pour apprendre à reconnaitre un élément dans une image, on fournit à la machine des images avec la présence de l'élément ou pas et un modèle est élaboré. Ce modèle permet à la machine, lorsqu'on lui fournit une image nouvelle, de dire si l'élément est présent ou pas. Ceci constitue la phase d'apprentissage. Ensuite, lorsque le modèle est suffisamment correct, on peut l'utiliser en poursuivant ou pas cet apprentissage. Pour poursuivre un apprentissage, il est nécessaire d'avoir un moyen de vérifier une réponse et de la valider, afin que la machine puisse prendre en compte dans son modèle les nouvelles informations.

On distingue trois grandes catégories d'apprentissage automatique :

- L'**apprentissage supervisé** qui comprend les algorithmes de classification permettant de classifier des objets tels que des images et les algorithmes de régression permettant 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. Le système exploite des exemples et acquiert la capacité à les généraliser ensuite sur de nouvelles données de production.

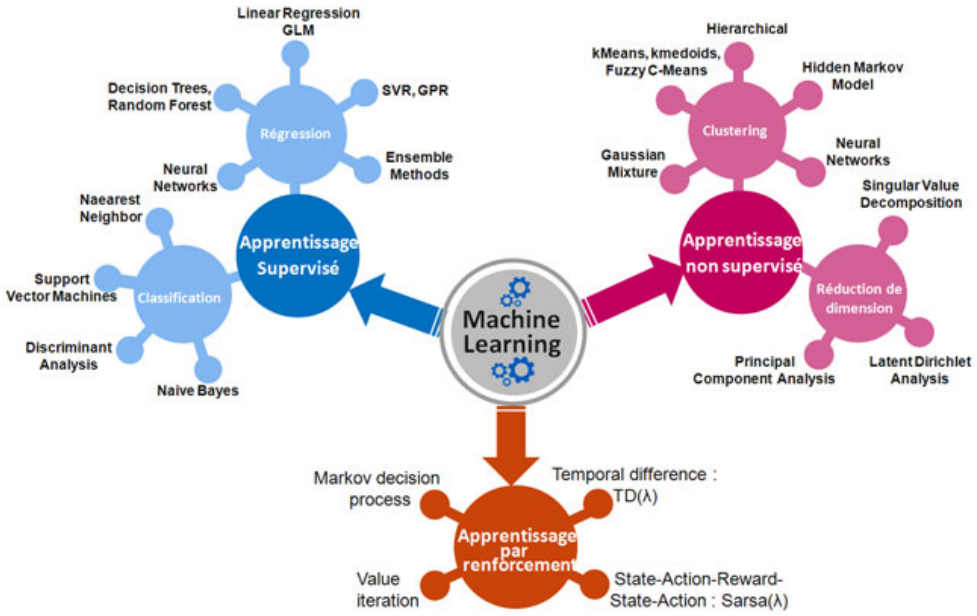

FIGURE 2 – Les trois types d'apprentissage.

- L'**apprentissage non supervisé** qui exploite des bases de données non labellisées. Ce n'est pas un équivalent de l'apprentissage supervisé qui serait automatique : les méthodes utilisées 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.

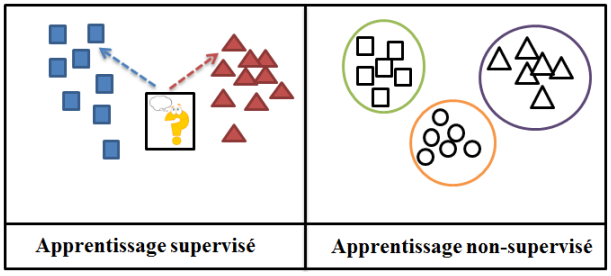

FIGURE 3 – Différences entre apprentissage supervisé et non supervisé.

- L'**apprentissage par renforcement** (*reinforcement learning* en anglais) permet d'ajuster des 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.L'apprentissage par renforcement s'appuie souvent sur les algorithmes profonds (*deep learning* en anglais) capables de mimer le comportement d'un réseau de neurones humain.

Nous nous intéresserons uniquement aux méthodes d'apprentissage supervisé et non supervisé qui sont les seules au programme.

1.2. Notion de partition

Les méthodes d'apprentissage que nous allons étudier nécessitent de classer une donnée parmi des données appartenant à un ensemble E. Il est nécessaire de disposer d'une classification. Chaque élé-

ment de E appartient à une et une seule classe. La question est de déterminer à quelle classe appartient la nouvelle donnée. Établir une classification consiste à construire une **partition** de E.

On dispose d'un ensemble E à n éléments avec une partition et nous souhaitons savoir à quelle partie appartient un élément de E. Si les parties sont représentées par des listes, un parcours séquentiel de chaque liste est nécessaire. Donc la complexité en temps est en $O(n)$.

Si on utilise une liste de listes comme `[[x1,5],[x2,2],...]` où un couple est constitué d'un élément de E et du numéro de la partie à laquelle il appartient, la complexité est encore linéaire en n. Une méthode efficace est d'utiliser une structure de dictionnaire. Les clés sont les éléments de l'ensemble E et les valeurs sont les numéros des parties : `x1:5,x2:2,...`. On obtient alors la partie à laquelle appartient un élément avec une complexité en temps constante.

2. Apprentissage non supervisé

2.1. Algorithme des k plus proches voisins

Un ensemble de données est connu et chacune des données appartient à une classe, ou une partie, bien déterminée. Cette classe est une des caractéristiques de chaque donnée et est liée aux autres caractéristiques.

Dans un apprentissage supervisé, l'algorithme doit permettre d'émettre une prévision sur cette caractéristique à propos d'une donnée dont on ne connaît que les autres caractéristiques, donc de prédire la partie dans laquelle la donnée peut être classée.

Une méthode consiste à baser cette prédiction sur la détermination des classes de ses k plus proches voisins, où k est à préciser, en retenant la classe majoritaire. On dispose donc d'un ensemble E de n points représentant des données classées ou étiquetées. L'ensemble E est l'**ensemble d'apprentissage**.

On choisit un entier k, plus petit que n, et on dispose d'un point x qui n'est pas dans E. Il s'agit de trouver parmi les points de E les k plus proches de x pour classer x. Le mot "proche" sous-entend une notion de distance. En pratique, nous utilisons la **distance euclidienne**, plus précisément le carré de la distance euclidienne comme dans la fonction définie ci-dessous :

```
def d(x,y) :
    n = len(x)
    d = 0
    for i in range(n) :
        d += x[i]**2 + y[i]**2
    return d
```

2.2. Exemple : le jeux de données Iris

Nous allons illustrer le principe de l'algorithme des k plus proches voisins sur un jeu de données très connu : le jeu de données *Iris* aussi connu sous le nom d'*Iris de Fischer*. Le jeu de données comprend 50 échantillons de chacune des trois espèces d'iris (*Iris setosa*, *Iris virginica* et *Iris versicolor*). Quatre caractéristiques ont été mesurées pour chaque échantillon : la longueur et la largeur des sépales et des pétales, exprimées en centimètres. Les premières lignes du jeux de données sont données ci-dessous :

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
Observation N°1	5.1	3.5	1.4	0.2
Observation N°2	4.9	3	1.4	0.2
Observation N°3	4.7	3.2	1.3	0.2
Observation N°4	4.6	3.1	1.5	0.2
Observation N°5	5	3.6	1.4	0.2
Observation N°6	5.4	3.9	1.7	0.4
Observation N°7	5	3.4	1.5	0.2
…	…	…	…	…
…	…	…	…	…

Dans la suite, nous nous intéresserons uniquement qu'à deux des quatre caractéristiques : la longueur des sépales et la largeur des pétales. Un élément est donc considéré comme un point dans un espace de dimension 2 représenté par une liste de deux flottants.

Nous allons découper le jeu de données en deux : un **jeu d'entrainement** et un **jeu de test**. Parmi les 150 échantillons de la base de données, 120 échantillons sont choisis aléatoirement pour entraîner notre algorithme, c'est le jeu d'entrainement, et les 30 autres sont gardés pour tester l'algorithme, c'est le jeu de test.

```
Data = pandas.read_csv("Iris.csv") # On crée un tableau de données
    contenant toutes les lignes et colonnes du jeu de données Iris
Train = Data.sample(frac=0.8,random_state=200) # On choisit
    aléatoirement 0.8*150=120 échantillons pour fabriquer le jeu
    d'entrainement
Test = Data.drop(Train.index) # les échantillons restants servent à
    fabriquer le jeu de test
```

La figure ci-après représente les trois types d'iris en fonction de la longueur des sépales et la largeur des pétales. Il semble que la connaissance de ces deux caractéristiques suffit pour prédire l'espèce d'un iris inconnu.

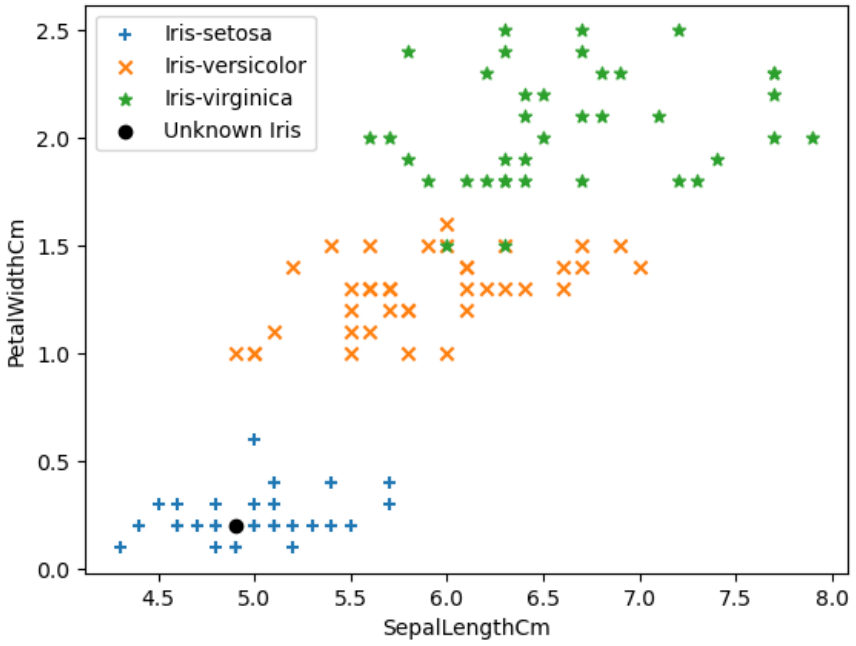

FIGURE 4 – Longueur des sépales et largeur des pétales en fonction de l'espèce de l'iris

Pour classifier l'iris inconnu, nous devons déterminer ces k plus proches voisins. Nous avons besoin au préalable définir une distance permettant de quantifier la proximité entre deux échantillons. Nous

utiliserons le carré de la distance euclidienne. Le code ci-dessous permet de calculer la distance entre deux points `X1` et `X2` dont les abscisses et ordonnées sont la longueur des sépales et la largeur des pétales

```
def d(X1,X2) :
    return( (X1[0]-X2[0])**2 + (X1[1]-X2[1])**2) # Calcul le carré de la
        distance euclidienne entre les points X1 et X2
```

Une fois définie la distance `d` sur l'ensemble étudié, nous pouvons déterminer les k pus proches voisins de l'échantillon P dont l'espèce est inconnue. La liste des points de E est triée suivant l'ordre croissant des distances au point P. On écrit donc une fonction de tri en utilisant la fonction `sorted` de Python.

```
def tri(E, P):
    def choix(elt): # pour trier suivant les valeurs d'indices 1
        return elt[1]
    distances = [(p, d(p, P)) for p in E]
    return sorted(distances, key=choix)
```

La fonction `knn` renvoie les k premiers points de la liste triée.

```
def knn(E, p, k):
    pts = tri(E, p)
    return [elt[0] for elt in pts[0:k]]
```

Il ne reste plus alors qu'à déterminer l'espèce prédominante parmi les k plus proches voisins de l'échantillon P.

```
def classe_maj(pts):
    Setosa = 0
    Versicolor = 0
    Virginica = 0
    for p in pts :
        if p[2]=='Iris-setosa':
            Setosa += 1
        elif p[2]=='Iris-versicolor':
            Versicolor += 1
        else :
            Virginica += 1
    if Setosa > Versicolor and Setosa > Virginica :
        return 'Iris-setosa'
    elif Setosa < Versicolor and Versicolor > Virginica :
        return 'Iris-versicolor'
    elif Setosa < Virginica and Versicolor < Virginica :
        return 'Iris-virginica'
    else :
        return 'unknown'
```

On choisit de se limiter au cinq plus proches voisins et on détermine l'espèce de l'iris inconnu P de la Figure 2 :

```
X = Train.loc[:,['SepalLengthCm','PetalWidthCm', 'Species']].values.
    tolist() # On crée une liste à partir des données du tableau Data
    créé précédemment
P = (Test['SepalLengthCm'].iloc[0], Test['PetalWidthCm'].iloc[0], Test['
    Species'].iloc[0]) # échantillon dont on veut déterminer l'espèce
    avec l'algorithme des k plus proches voisins
classe_maj(knn(X, P, 5))
```

L'algorithme retourne les cinq voisins suivants :

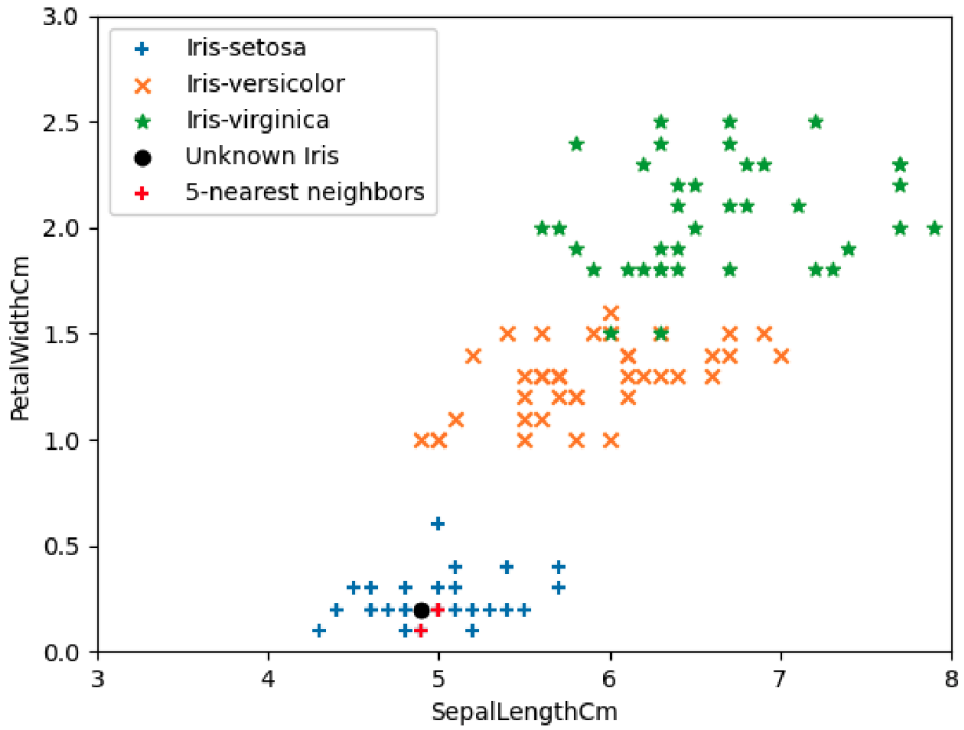

FIGURE 5 – Les cinq plus proches voisins de l'iris inconnu.

```
[[5.0, 0.2, 'Iris-setosa'],
 [5.0, 0.2, 'Iris-setosa'],
 [5.0, 0.2, 'Iris-setosa'],
 [4.9, 0.1, 'Iris-setosa'],
 [4.9, 0.1, 'Iris-setosa']]
```

Les cinq plus proches voisins appartenant tous à l'espèce *setosa*, l'algorithme classe l'échantillon P dans les iris *setosa*. Il s'agit bien de l'espèce de cet iris. On peut répéter cette opération pour les trente iris du jeu de test : l'algorithme attribue la bonne espèce à 26 d'entre eux et se trompe pour seulement quatre d'entre eux.

La notion de plus proches voisins est présente dans de nombreux domaines. Par exemple, elle est utilisée dans les moteurs de recherche pour faire des recommandations automatiques ou dans le domaine médical pour faire des prédictions de crises cardiaques ou de certaines maladies.

Estimons la complexité temporelle de cet algorithme. Le calcul des distances est en $O(n)$ et le calcul de la classe majoritaire parmi les k plus proches voisins est de complexité constante. Le point bloquant est le calcul de k voisins qui implique une étape de tri, dont la complexité dépendra de l'algorithme de tri choisi et fixera celle de l'algorithme principal. ici, nous avons utiliser la fonction `sorted` de Python, qui est de complexité quasi-linéaire en n selon la documentation en ligne.

Remarque – Pour chaque prédiction, on est obligé de parcourir toute la base d'apprentissage E. Lorsque cette dernière est très grande (on peut avoir jusqu'à $n = 1$ million de données d'apprentissage !), il devient très coûteux de garder en mémoire ces n données et de les tester exhaustivement à chaque nouvelle observation pour déterminer ses k voisins les plus proches, ce qui rend cet algorithme peu efficace par rapport aux autres algorithmes d'apprentissage automatique lorsqu'il est confronté à des problèmes de grande taille.

Remarque – L'algorithme des k plus proches voisins dans le cadre d'une classification, mais il est facile de l'adapter pour résoudre un problème de régression : au lieu de renvoyer la classe majoritaire parmi les k voisins les plus proches du nouvel élément, on renvoie simplement la moyenne des cibles y_i de ses k voisins.

L'algorithme des k plus proches voisins permet d'émettre des prédictions. Il reste à mesurer la qualité de ces prédictions.

2.3. Matrice de confusion

Considérons un test censé prédire si une personne est porteuse ou non d'une maladie. Ce test fait la bonne prédiction dans 90 pour cent des cas. Cela signifie que dans 90 pour cent des cas des cas, si une personne est porteuse de la maladie le test est positif et sinon le test est négatif. On parle de "**vrais positifs**" et de "**vrais négatifs**". Dans 10 pour cent des cas le test ne fait pas la bonne prédiction. Le résultat est donc soit positif alors que la personne n'est pas porteuse de la maladie, soit négatif alors que la personne est porteuse. On parle alors de "**faux positifs**" et de "**faux négatifs**".

Il est bien sûr intéressant de connaître la répartition de ces quatre types de résultat. Revenons sur l'exemple précédent des iris. On peut classifier de la manière suivante les prédictions obtenues :

Espèce prédite / Espèce réelle	Verosa	Versicolor	Virginica
Verosa	8	0	0
Versicolor	0	8	2
Virginica	0	2	10

A partir de ces données, on peut obtenir la **matrice de confusion** définie par : $\begin{pmatrix} 8 & 0 & 0 \\ 0 & 8 & 2 \\ 0 & 2 & 10 \end{pmatrix}$.

Cette matrice permet d'obtenir des informations sur la qualité des prédictions. On peut, avec cet outil, mesurer le niveau de pertinence d'un modèle ou son degré de confusion. En particulier, on peut connaître le taux d'erreurs commises dans des prédictions et le type de ces erreurs. On peut par exemple définir :

- Le **taux d'erreurs** TE : c'est le nombre de prédictions incorrectes sur le nombre total de prédictions. On vise une valeur la plus proche possible de 0 ;
- La **précision** P : c'est le nombre de prédictions correctes sur le nombre total de prédictions. On vise une valeur la plus proche possible de 1. On a la relation : $P = 1 - TE$;

Remarque – Le choix de la valeur de k est important. Si k est trop petit, le modèle est sensible au bruit des données. Si un iris est proche d'une valeur aberrante, la classe renvoyée sera incorrecte. À l'inverse si k est trop grand, la classe renvoyée est la classe dominante dans le jeu de données. On donne dans le table ci-dessous le nombre de prédictions correctes en fonction de la valeur de k choisie. Pour cet exemple, il semble préférable de choisir une valeur de k comprises en 3 et 5.

Nombre de plus proches voisins k	1	2	3	4	5	6	10
Nombre de prédictions correctes	23	21	26	26	26	25	24

Le choix de la classe majoritaire parmi les voisins ne donne pas toujours satisfaction. Différents facteurs entre en jeu comme la distribution des classes. Une amélioration consiste parfois à pondérer les classes des voisins à l'aide des distances entre ces voisins et le point à classer.

3. Apprentissage supervisé

3.1. Algorithme des k-moyennes

L'*algorithme des k-moyennes* est l'un des algorithmes de classification les plus répandus. En anglais on parle de *clustering algorithm*, *cluster* signifiant groupe.L'algorithme consiste à regrouper des données possédant des éléments de similarité dans des groupes.

À partir d'un jeu de données, on définit un nombre k de groupes ou classes. L'algorithme permet d'analyser le jeu de données et d'affecter chaque donnée à une classe, des données similaires ou proches appartenant alors à une même classe. Dans un apprentissage supervisé, on essaie de trouver une corrélation entre les valeurs des données d'un ensemble et une valeur à prédire. Ici on essaie de trouver des ressemblances entre les données pour constituer une partition. Pour déterminer la

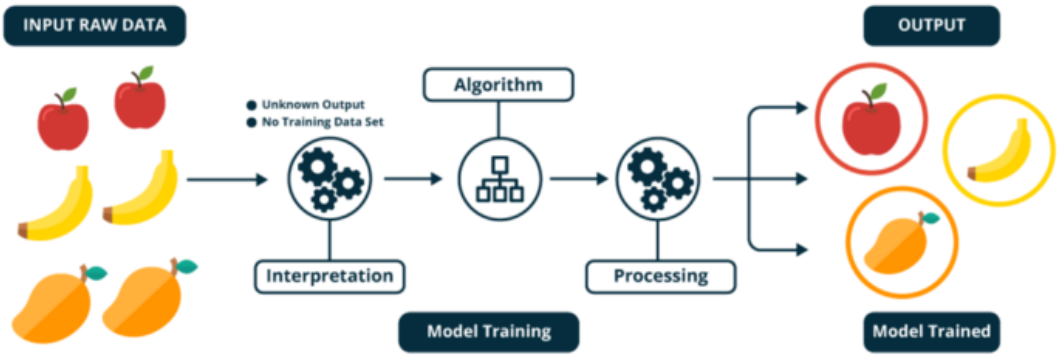

FIGURE 6 – Apprentissage non supervisé.

proximité entre deux données on utilise une mesure de similarité ou une distance. Deux données sont considérées comme proches lorsque les éléments qui permettent de décrire les données ont des valeurs proches. Là encore, nous utilisons la distance euclidienne :

$$d(X,Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

L'objectif est de partitionner l'ensemble des données en groupes distincts, chaque groupe étant représenté par une donnée centrale dont les autres données du groupe sont proches. Ces centres peuvent être choisis au hasard parmi toutes les données à l'initialisation. Chaque itération consiste à affiner le partitionnement. Pour cela le centre de chacun des k groupes est ré-évalué par le calcul du barycentre de l'ensemble des données du groupe. C'est de là que vient le nom d'algorithme des k-moyennes, puisqu'à chaque itération, k moyennes sont calculées. Les données sont alors regroupées en utilisant les nouveaux centres.

Remarque – Les centres calculés à chaque itération ne sont plus nécessairement des éléments de l'ensemble de données.

Algorithme des k-moyennes

Un ensemble de données est fourni sous forme de matrice, le nombre de groupes k est choisi.

- Initialiser de manière aléatoire les centres de chaque groupe.
- Répéter les deux étapes :
  - regrouper chaque donnée dans la partie définie par le centre le plus proche ;
  - remplacer chaque centre par le barycentre des données de son groupe.

L'algorithme se termine soit au bout d'un nombre d'itérations fixé à l'avance soit lorsque que celui-ci à converger, c'est-à-dire lorsque les centres ne sont plus modifiés lors d'une itération et que l'algorithme a alors trouvé une partition stable de l'ensemble de données. Le choix du nombre de groupes k est important. Sans hypothèse préalable sur les données, si leur nombre est important, il est nécessaire de tester plusieurs valeurs. Si k est trop petit, les données dans chaque groupe risque de ne pas avoir une similarité forte, et si k est trop grand chaque groupe ne permet pas de découvrir des caractéristiques intéressantes. L'algorithme des k-moyennes a de nombreux champs d'application : détection de valeurs aberrantes, regroupement d'images, data marketing, etc.

3.2. Exemple : classification des *Iris de Fischer*

Reprenons l'exemple du jeu de données des *Iris de Fischer*. Nous supposerons dorénavant que nous ne connaissons pas les espèces des différents échantillons. Nous allons chercher à séparer les échantillons en k groupes en nous basant sur les caractéristiques utilisées précédemment : la longueur des sépales et la largeur des pétales.

```
#Conversion des données sous forme de liste
Train = Data.loc[:,['SepalLengthCm','PetalWidthCm']].values.tolist() #
    On ne possède plus l'information sur l'espèce.
```

Pour écrire l'algorithme des k-moyennes, nous aurons besoin de plusieurs fonctions auxiliaires.
Nous allons d'abord écrire une première fonction permettant de choisir aléatoirement k centres parmi tous les échantillons du jeu de données.

```
def centres_initiaux(E,k):
    n = len(E)
    liste = [] # liste qui contiendra les indices des k centres choisis
    for i in range(k):
        x = rd.randrange(n) # Génère un entier aléatoirement compris
            entre 0 et n-1 en utilisant la bibliothèque random
        while x in liste : # Si le point a déjà été pris comme centre
            (présent dans liste) on génère un nouveau centre
            x = rd.randrange(n)
        liste.append(x)
    return liste
```

Nous aurons également besoin d'une fonction `barycentre` permettant de calculer l'isobarycentre d'un ensemble de points et d'une fonction `ind_minimum` permettant de renvoyer l'indice du centre le plus proche d'un points de l'ensemble.

```
def barycentre(points):
    n = len(points) # nombre de points
    m = len(points[0]) # nombre de coordonnées de chaque point
    s = [0] * m # coordonnées du barycentre
    for p in points:
        for k in range(m):
            s[k] = s[k] + p[k]
    s = [s[k]/n for k in range(m)]
    return s
```

```
def ind_minimum(dist):
    mini = min(dist)
    choix = []
    for i in range(len(dist)):
        if dist [i]== mini:
            choix.append(i)
    return choix
```

Nous pouvons alors écrire une fonction `partition_initiale` qui permet de proposer une première partition de l'ensemble des données à partir de k centres choisis aléatoirement et de calculer les nouveaux centres des groupes obtenus.

```
def partition_initiale(E, centres):
    n = len(E)
    k = len(centres)
    partition =[[c] for c in centres]
    classes = { centres[i] : i for i in range(k)}
    # Classification des points dans les différents clusters
    for i in range(n) :
        if i not in centres:
            dist = [d(E[i],E[k]) for k in centres ] # Calcul la distance
                du point i à chacun des centres
            choix = ind_minimum(dist)[0] # indice du centre le + proche
            classes[i] = choix # le point i est dans le cluster d'indice
                choix
            partition[choix].append(i) # On ajoute le point i au cluster
                choix
    # Calcul des nouveaux centres de clusters
```

```
    nouveaux_centres = [None] * k
    for i in range(k):
        points = [E[j] for j in partition[i]]
        nouveaux_centres[i] = barycentre(points)
    return partition, nouveaux_centres, classes
```

Pour $k = 3$, les trois groupes construits à partir de trois centres choisis aléatoirement sont représentés sur la figure ci-dessous.

```
Centres_initiaux = centres_initiaux(Train,3)
Partition, Nouveaux_centres, Classes = partition_initiale(Train,
    Centres_initiaux)
```

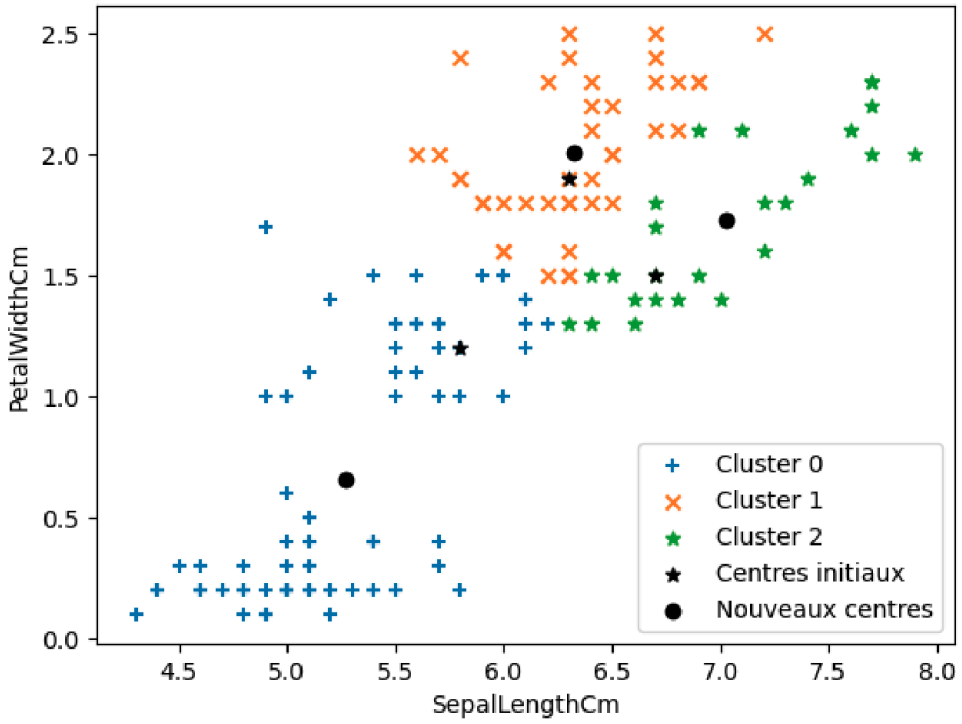

FIGURE 7 – Groupes obtenus à partir de trois centres choisis aléatoirement.

Nous pouvons alors écrire la fonction `k_means`.

```
def k_means(E, centres, max_iter) :
    # Initialisation de la classification
    n = len(E)
    k = len(centres)
    partition, centres, classes = partition_initiale(Train,centres) # On
        crée la premièère partition
    # Itérations
    iteration = 0
    evolution = True # booleen permattant de tester si l'algorithme à
        converger
    while evolution and iteration < max_iter :
        evolution = False
        iteration += 1
        for i in range(n):
            dist = [d(E[i], centre) for centre in centres] # Calcul la
                distance du point i à chacun des centres
            choix = ind_minimum(dist)[0] # indice du centre le + proche
            # Modification si changement de cluster
            if choix != classes[i] :
                partition[classes[i]].remove(i) # On enlève le point i
                    de l'ancien cluster
```

```
                classes[i] = choix # On attribue le bon cluster à i
                partition[choix].append(i) # On ajoute le point i au bon
                    cluster
                evolution = True
        # Calcul des nouveaux centres
        for i in range(k) :
            points = [E[j] for j in partition[i]]
            centres[i] = barycentre(points)
    return partition, centres
```

En appliquant l'algorithme des k-moyennes aux *Iris de Fischer*, on obtient la classification suivante.

```
Centres_initiaux = centres_initiaux(Train,3)
partition, centres = k_means(Train, Centres_initiaux, 50)
```

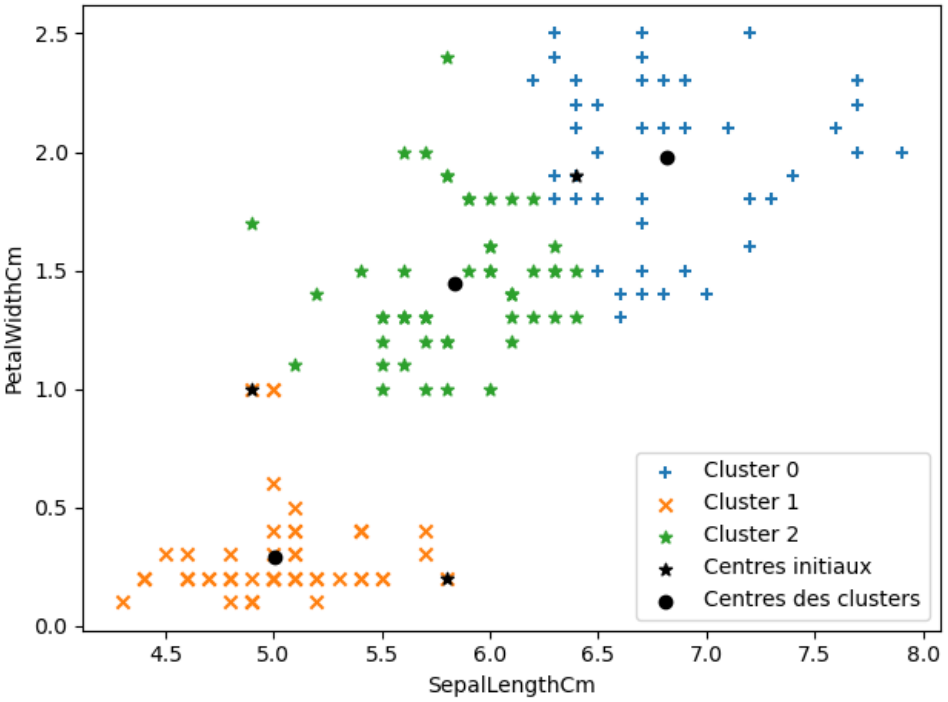

FIGURE 8 – Groupes obtenus après convergence de l'algorithme des k-moyennes.

On remarquera que les groupes obtenus ne correspond pas au groupes obtenus en se basant sur les espèces d'iris.

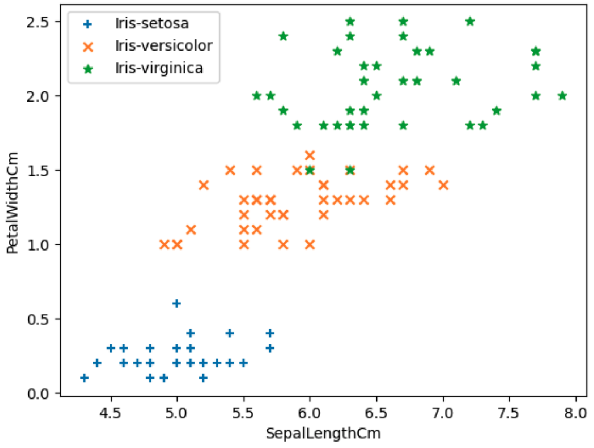

FIGURE 9 – Groupes obtenus à partir des espèces.

3.3. Convergence vers des minima locaux

On peut démontrer qu'à chaque itération, la variance intra-classe diminue. Comme il n'y a qu'un nombre fini de façons de former les différents groupes avec les N données, on en déduit que l'algorithme termine et que la configuration atteinte est un **minimum local**. Toutefois, le nombre de façons de former les différents groupes est certes fini, mais grand, et le minimum atteint n'est pas nécessairement **global** : nous n'avons pas de garanties théoriques sur la qualité de la solution obtenue, ni sur le temps mis pour y parvenir. En pratique, l'algorithme se termine assez rapidement. Pour l'exemple des *Iris de Fischer*, quelques soit les centres initialement choisis, l'algorithme converge en moins d'une dizaine d'itérations. L'initialisation des centres affecte souvent beaucoup le résultat final. Si on exécute deux fois l'algorithme sur les mêmes données pour des centres initiaux différents, on n'obtiendra pas nécessairement le même partitionnement à la fin. Pour palier à ce problème, une stratégie consiste à comparer les groupes obtenus pour différents centres initiaux choisis aléatoirement. Dans le cas d'exemple des *Iris de Fischer*, quelque soit les centres choisis initialement l'algorithme converge vers les mêmes groupes : la solution obtenue est *a priori* un minimum global.

4. Exercice

Inspiré par le roman 1984 de George Orwell, un pays décide d'instaurer une police de la pensée afin de tenir en bride ses citoyens. Pour pouvoir intervenir plus efficacement, ses fonctionnaires décident de créer un algorithme d'IA de surveillance massive qu'ils appellent affectueusement *Big Brother*.

4.1. Question préliminaire

1. Écrire une fonction `dist(x1,x2)` qui prend en arguments deux listes de même taille, et calcule la distance euclidienne entre les deux.

4.2. Note d'insoumission (apprentissage supervisé)

On s'intéresse à la manière dont *Big Brother* établit les notes d'insoumission des citoyens. La police a sélectionné aléatoirement N citoyens du pays, et a attribué à chaque individu une note d'insoumission en se basant sur les délits commis (plus la note est élevée, plus la personne est susceptible de se rebeller contre l'autorité).

On suppose que les données criminelles des N individus sont stockées sous forme numérique dans un `array` appelé délits de taille $N \times p$, la i-ième ligne correspondant aux caractéristiques x_i de l'individu i (i est donc compris entre 0 et $N-1$). On disposera également d'une liste notes de longueur N, où la i-ième case contient la note d'insoumission attribuée à l'individu i par la police.

Il s'agit de programmer l'algorithme des k plus proches voisins pour déterminer les notes d'insoumission du reste de la population. On appellera observation un vecteur de taille p représentant les données criminelles d'un individu.

2. Écrire une fonction `distances_au_point(delits, x )` qui prend en arguments la matrice contenant les données criminelles des N individus et une liste de longueur p contenant les données criminelles d'un nouvel individu, et qui renvoie une liste distances de longueur N où la i-ème case contient la distance de x à l'observation i de délits.

3. Écrire une fonction `trier_individus(distances)` qui prend en argument une liste contenant les distances de x à chaque observation et qui renvoie une liste `indices_tries` contenant les indices de la liste rangés par distance croissante. On pourra utiliser la fonction `sorted` de Python.

Exemple – Si `distances = [5,9,1]`, alors la fonction `trier_individus(distances)` renvoie `[2,0,1]` (la case d'indice 2 présente la distance la plus petite, puis c'est la case d'indice 0, et enfin la case d'indice 1).

Remarque – La fonction `sorted` a un paramètre `key` qui spécifie une fonction qui va être appliquée sur chaque élément de la liste. Le tri se fait alors dans l'ordre croissant des sorties de la fonction. Ici, le paramètre `key` correspond à une fonction $f(i)$ qui prend en entrée un indice i et qui renvoie distances `[i]`.

4. Écrire une fonction `moyenne_des_k_voisins(indices_tries, notes, k)` qui prend en arguments la liste contenant les indices des observations triées par distance croissante à x, la liste contenant les notes d'insoumission des individus et le nombre de voisins à considérer, et qui renvoie la moyenne des notes d'insoumission des k plus proches voisins de x.

5. En déduire une fonction `k_plus_proches_voisins(delits, notes, x,k )` qui renvoie la note d'insoumission estimée pour le citoyen ayant x comme données criminelles.

4.3. Groupes de surveillance (apprentissage non supervisé)

Afin de faciliter la tâche aux agences de renseignement, *Big Brother* cherche à répartir les citoyens en plusieurs groupes homogènes. Pour cela, il dispose de nombreuses informations personnelles sur chaque citoyen ; à partir de ces descripteurs, il classera l'individu dans un des groupes, ce qui déterminera le niveau de surveillance à lui appliquer.

On suppose que ces informations sont stockées sous forme numérique dans un `array` appelé `infos` de taille $M \times q$, la i-ième ligne correspondant aux informations personnelles de l'individu i (où i est compris entre 0 et $M-1$).

Le but est de répartir ces M citoyens dans k groupes homogènes. L'algorithme des k-moyennes consiste à fixer k pôles aléatoirement, à étiqueter chaque individu selon le pôle le plus proche, de recalculer les valeurs des pôles comme barycentres des groupes, et de recommencer jusqu'à ce que les pôles ne bougent plus. Les caractéristiques des pôles seront stockées dans un `array` appelé `poles` de taille $k \times q$.

6. Pour fixer les valeurs initiales des k pôles, on procède de la manière suivante : pour chacun des k pôles, pour chacun de ses q descripteurs, on choisit une valeur aléatoire uniformément répartie entre la valeur minimale et la valeur maximale de ce descripteur calculée sur l'ensemble des M données. Écrire une fonction `initialisation_poles(infos, k)` qui génère la matrice `pole` initiale. On pourra utiliser les fonctions `min` et `max` de Python, ainsi que la fonction `uniform(a,b)` du package `random`, qui renvoie un nombre aléatoire compris entre a et b.

7. Après l'étape d'initialisation, il faut affecter chaque observation à son pôle le plus proche. Écrire une fonction `pole_le_plus_proche(poles, x)` qui prend en arguments la matrice contenant les caractéristiques des pôles et une liste contenant les caractéristiques d'un individu, et qui renvoie l'indice du pôle le plus proche de x.

8. Écrire une fonction `poles_les_plus_proches(poles, infos)` qui renvoie une liste qu'on appellera `repartition`, dont la i-ième case contient l'indice du pôle le plus proche de l'observation i.

9. La troisième étape consiste à recalculer les valeurs des pôles pour les placer au barycentre de leur catégorie. Autrement dit, on affecte au pôle d'indice ℓ la moyenne des observations qui ont été catégorisées comme étant de classe ℓ.

Écrire une fonction `nouveaux_poles(infos, repartition, k)` qui renvoie une matrice `poles` contenant les caractéristiques des barycentres de chaque catégorie.

10. En déduire une fonction `k_moyennes(infos, k)` qui renvoie la matrice `poles` des pôles, correspondant aux k classes trouvées par l'algorithme des k-moyennes.

11. Si un nouveau citoyen atteint la majorité, comment le classerons-nous ? On suppose qu'on dispose de ses données complètes, rangées dans une liste x de longueur q, et des caractéristiques `poles` des pôles calculées par l'algorithme des k-moyennes.