Intelligence artificielle et jeux

Liste des exercices

Exercice 1 Achat d'une voiture ★

Une entreprise de voiture mène une campagne publicitaire et souhaite cibler le public le plus enclin à effectuer un tel achat. À l'issu de cette étude, on remarque que les critères décisifs pour déterminer si une personne est intéressée est son âge, ainsi que son revenu annuel. Les données sont extraites dans le fichier

`achat_voiture.csv`

On notera N le nombre total d'individus dans la base de données.

1. Téléchargez ce fichier et créez un programme python permettant de le traduire sous la forme d'un tableau `numpy` :

```python
import numpy as np

path = 'Chemin\\vers\\le\\fichier\\achat_voiture.csv'
dataset = np.genfromtxt(path, delimiter=',', dtype=float)[1:]

```

2. Ajoutez ensuite trois lignes permettant de définir les tableaux suivants (chacuns de :
   - `ages` (taille N) contient les ages des individus ;
   - `salaires` (taille N) contient les salaires des individus (en k€) ;
   - `achats` (taille N), une liste de booléens, indiquant si un individu a effectivement acheté une voiture ou non.

On va écrire une fonction `achat(age, salaire, nombre_voisins = 3)`, prenant en entrée les caractéristiques d'un individu (son âge et son salaire), et renvoyant la prédiction du modèle quant à un potentiel achat, en suivant un algorithme des k plus proches voisins.

3. Écrire dans cette fonction, une ligne calculant un tableau `distances` (taille N) stockant pour chaque personne de la base de donnée, sa distance à l'individu étudié.
   NB : On utilisera pour cela la distance euclidienne, définie dans un espace à deux dimensions comme :

$$\begin{cases} p_1 = (x_1, y_1) \\ p_2 = (x_2, y_2) \end{cases} \implies d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

4. Il faut ensuite trouver les plus proches voisins de l'individu test. Pour cela, recopiez et complétez le code ci-dessous, de sorte que la liste `voisins` contienne les indices des personnes les plus proches de celle qu'on étudie :

```python
voisins = []
for i in range(??):
    index = distances.argmin()  # la fonction argmin renvoie l'indice du plus petit élément de la liste
    voisins.append(??)
    distances[index] = ??

```

5. Ajouter une dernière ligne renvoyant la probabilité que l'individu test achète une voiture. Cette probabilité sera définie comme la fraction de ses plus proches voisins ayant effectué un tel achat.
   Ex : Si 3 voisins parmi les 5 plus proches ont acheté une voiture, la fonction doit renvoyer

$$\frac{3}{5} = 60\%$$

Pour vérifier que votre code fonctionne correctement, vous pouvez ajouter les lignes suivantes à votre fonction :

```
plt.scatter(ages[achats == 1], salaires[achats == 1], color='r', alpha=.1, label='Achat')
plt.scatter(ages[achats == 0], salaires[achats == 0], color='b', alpha=.1, label="Pas d'achat")
plt.xlabel('Âge')
plt.ylabel('Salaire')
plt.legend()

for voisin in voisins:
    plt.plot([age, ages[voisin]], [salaire, salaires[voisin]], color=['b', 'r'][int(achats[voisin])])
plt.scatter(age, salaire, color=['b', 'r'][int(achat)])

plt.show()

```

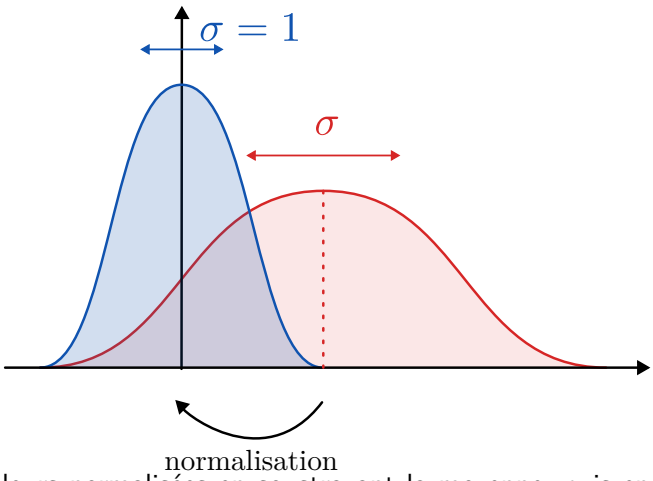
Il faudra bien sûr veiller à importer le module `pyplot` :

```
import matplotlib.pyplot as plt
```

Ces lignes ont pour but d'afficher les données d'entraînement et de positionner l'individu ciblé, en faisant apparaître ses plus proches voisins.

6. Testez votre fonction `achat` en faisant varier les arguments utilisés et vérifiez qu'elle fonctionne correctement.
7. Modifiez la ligne définissant `salaires`, de manière à finalement garder une expression en €. Refaire les tests précédents. Quel problème majeur constate-t-on ?

Pour pallier à ce problème, il faut en effectuer le calcul de `distances` sur les données **normalisées**. C'est-à-dire ramenées arbitrairement à une moyenne nulle et un écart-type de 1 :



Si l'on dispose d'un jeu de données x, on peut construire les valeurs normalisées en soustrayant la moyenne, puis en divisant par l'écart-type :

$$x_{\text{norm}} = \frac{x - \langle x \rangle}{\sigma(x)}$$

8. a) Avant le calcul des distances, définissez les valeurs suivantes `moyenne_ages`, `sigma_ages`, `moyenne_salaires` et `sigma_salaires` (les moyennes et écart-types de chacune des listes de données).
   b) Servez-vous de ces valeurs pour créer les listes normalisées `ages_norm` et `salaires_norm`. Faire de même pour en construisant `age_norm` et `salaire_norm`, les versions redimensionnées des paramètres `age` et `salaire`.
   c) Modifiez ce qu'il faut en conséquence dans la suite du code.
   d) Testez à nouveau votre fonction, et vérifiez que le problème évoqué question 7 est réglé.

Exercice 2 Prédiction de la potabilité d'une eau ★★

Une organisation publique cherche à anticiper le caractère potable ou non d'une eau, à partir des mesures chimiques. Elle choisit donc d'utiliser un algorithme d'intelligence artificielle, dont la phase d'apprentissage a permis de répertorier plusieurs eaux, enregistrées dans le fichier

`water_potability.csv`

1. Téléchargez ce fichier et créez un programme python permettant de le traduire sous la forme d'un tableau `numpy` :

```
import numpy as np

path = 'Chemin\\vers\\le\\fichier\\water_potability.csv'
columns = np.genfromtxt(path, delimiter=',', dtype=str)[0]
dataset = np.genfromtxt(path, delimiter=',', dtype=float)[1:]

```

Remarque :

- La variable `columns` stocke les noms des mesures chimiques effectuées pour chacune des eaux.
- La variable `dataset` stocke les valeurs mesurées et indique en dernière colonne si l'eau est potable (valeur 1) ou non (valeur 0).

Le but est de créer une fonction `potabilite(x, nombre _voisins = 3)` de la forme :

```
def potabilite(x, nombre_voisins = 3):
    '''
    Arguments:
    - x est un tableau contenant les caractéristiques de l'eau étudiée
    - nombre_voisins indique le nombre de plus proches voisins à prendre en compte
    Cette fonction renvoie un booléen indiquant si l'eau caractérisée par les mesures x est potable ou non
    '''

    # 1- Extraction des données d'entraînement
    # 2- Normalisation des données d'entraînement
    # 3- Calcul des distances
    # 4- Recherche des plus proches voisins
    # 5- Évaluation de la potabilité de l'eau

```

2. S'agit-il d'un problème de régression ou de classification ?

3. **Extraction des données d'entraînement :**
   Écrire deux lignes permettant de définir les tableaux `X` et `Y` de la manière suivante :
   - `X` contient une ligne par eau testée et stocke en colonnes toutes les valeurs d'entrée (mesures chimiques) ;
   - `Y` est une liste de booléens, contenant pour chaque eau son caractère potable.
4. **Normalisation des données d'entraînement :**
   De la même manière que dans l'exercice 1, il faut normaliser les valeurs. Pour cela on utilisera un programme ayant la structure suivante :

```
    xn = np.zeros((len(x))) # Construit une liste remplie de 0, de même taille que x
    Xn = np.zeros(X.shape)  # Construit un tableau de 0, de mêmes dimensions que X
```

```
    for k in range(X.shape[1]): # On parcourt les colonnes
        mu =          # Calcul de la moyenne de cette colonne pour toutes les eaux
        sigma =       # Calcul de l'écart-type
        xn[k] =       # Affectation de la valeur normalisée pour x
        Xn[:, k] =    # Affectation de la valeur normalisée pour X

```

En vous aidant des méthodes `mean` et `std` (cf. annexe), recopier et compléter ces lignes afin de définir correctement `xn` et `Xn`.

5. **Calcul des distances :**

   À l'aide de la méthode `sum` (cf. annexe), ajouter une ligne permettant de calculer les distances euclidiennes entre l'eau étudiée (caractérisée par `x`) et chaque eau des données d'entraînement (lignes de `X`).

6. **Recherche des plus proches voisins et évaluation de la potabilité de l'eau :**

   En vous inspirant de l'exercice 1 terminer la fonction `potabilite`, de sorte qu'elle identifie les plus proches voisins de l'eau étudiée, et qu'elle renvoie le résultat quant à sa potabilité.

Test :

Pour tester votre fonction, vous pouvez également lui faire afficher les indices des voisins. Voici ce que vous devriez obtenir pour `nombre_voisins = 5`, dans différents cas :

```
>>> potabilite([7.03,179,28827,4.9,389,593,12.0,58.3,4.36], 5)
[255, 260, 211, 271, 105]
>>> potabilite([5.94,184,17527,5.3,339,409,15.2,31.9,3.27], 5)
[312, 73, 34, 225, 18]
>>> potabilite([8.15,158,19663,8.4,283,431,20.6,59.9,4.40], 5)
[251, 171, 59, 159, 308]
```

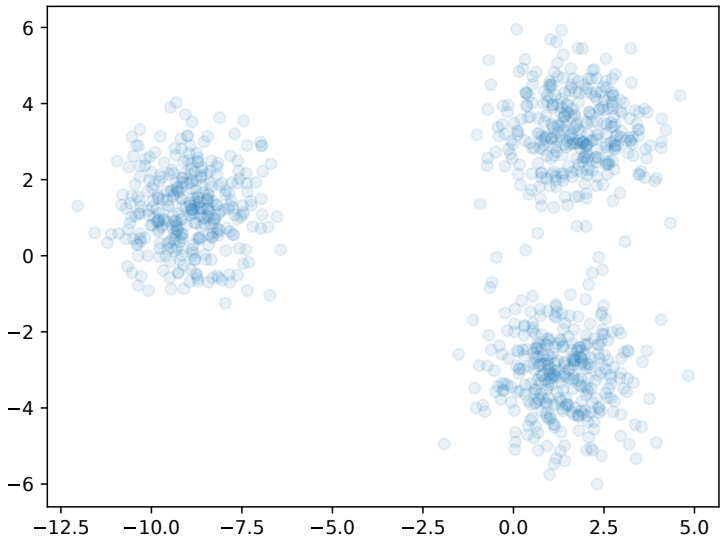
Exercice 3 Algorithme des k-moyennes ★★

Dans cet exercice, on aura besoin d'importer les bibliothèques suivantes :

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import numpy as np
```

La fonction `make_blobs` permet de construire un échantillonnage de plusieurs points agrégés en paquets (blobs).

```
>>> X, classes = make_blobs(1000, 2, 3)
>>> X   # Tableau de 1000 points contenant 2 coordonnées, agrégés en 3 blobs
array([[ 8.01538049,  1.68443094],
       [ 8.55593467, -2.03039211],
       [-0.41762477, -2.91395026],
       ...,
       [-0.4658861 , -3.59519554],
       [ 6.75717502, -1.33999468],
       [-0.64828968, -5.90617413]])
>>> plt.scatter(X[:, 0], X[:, 1], alpha = .1)   # alpha contrôle l'opacité
>>> plt.show()
```



Le but de cet exercice est de coder une fonction `Kmean(X, n_clusters)` utilisant un algorithme des k-moyennes pour trier les données `X` en un nombre `n_clusters` de groupes (appelés *clusters*). Cette fonction se présentera de la manière suivante :

```
def Kmean(X, n_clusters, N=10):

    barycentres = ...       # Initialisation des positions des barycentres

    for k in range(N):  # On pourra changer le nombre de répétitions

        clusters = # Initialisation des clusters de points
        # Remplissage des clusters
        # Calcul des nouvelles positions des barycentres

    # Affichage des points et des barycentres
```

1. En vous servant de la fonction `np.random.uniform`, écrire la ligne permettant d'initialiser les barycentres des clusters à des positions aléatoires.
   Indication : on peut donner trois arguments importants à cette fonction `low`, `high` et `size`. Voici un exemple d'utilisation :

```
>>> np.random.uniform(low = 0, high = 10, size = (3, 2))
array([[3.58376299, 9.74265923],
       [0.22802474, 5.67105891],
       [4.85588259, 0.62212047]])
```

2. Écrire la ligne d'initialisation des groupes de points. La variable `clusters` doit être une liste de listes (initialement vides). Chaque élément représentera un cluster de points.
3. Pour remplir les clusters, il faudra déterminer pour un point donné, le barycentre le plus proche. Cette question vise à créer une fonction `get_closest_barycentre(point, barycentres)` prenant en entrée un point et la liste des barycentres de clusters identifiés et renvoie l'indice du barycentre le plus proche.

   Si vous vous sentez à l'aise, tenter de créer une telle fonction. Sinon, aidez-vous des étapes suivantes :

   - Le but d'une telle fonction est de stocker dans une variable `distance_min` la distance minimale entre le point et un barycentre. Comment initialiser cette grandeur ?
   - De même, il faut initialiser une variable `cluster_index`, que l'on cherchera à renvoyer à la fin de la fonction.
   - Pour chaque barycentre, calculer la distance (euclidienne) qui le sépare du point considéré. Mettre à jour `distance_min` et `cluster_index` **s'il le faut**.
   - Renvoyer `cluster_index`.
4. Dans la fonction `Kmean`, ajouter quelques lignes permettant de remplir les clusters, en vous servant de `get_closest_barycentre`.
5. Il faut à présent coder les nouvelles positions des barycentres de chaque cluster. Si un cluster est vide, c'est que le barycentre associé est trop loin des points. Dans ce cas on le réinitialisera avec une valeur aléatoire.
6. Utiliser la fonction `plt.scatter` pour afficher les points créés et vérifier que les barycentres représentent correctement les groupes identifiables.

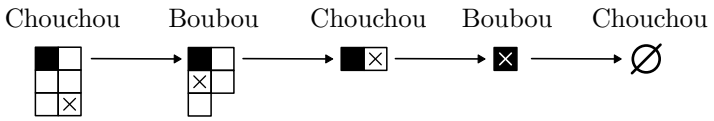
Exercice 4 Tablette de chocolat empoisonnée (Méthode des attracteurs) ★★★

Chouchou et Boubou veulent se partager un tablette de chocolat, mais le carreau en haut à gauche est empoisonné ! Ils jouent chacun à leur tour selon les règles suivantes :

- le premier joueur choisit un carreau encore disponible ;
- il retire alors ce carreau ainsi que tous ceux présents en bas et/ou à droite de celui-ci ;
- c'est au tour du joueur suivant de jouer selon les mêmes règles ;
- le joueur qui retire le dernier carreau a perdu !

Exemple

On peut par exemple imaginer la partie suivante sur une tablette (3×2) :



Explications :

- Chouchou commence en choisissant le carreau en bas à droite et retire donc uniquement un carré ;
- Boubou choisit ensuite le premier carreau de la deuxième ligne, ce qui enlève 3 carrés du jeu ;
- Chouchou n'a pas le choix, il doit choisir le seul carré non empoisonné ;
- Il ne reste qu'un carré (empoisonné) pour Boubou, qui perd donc la partie !

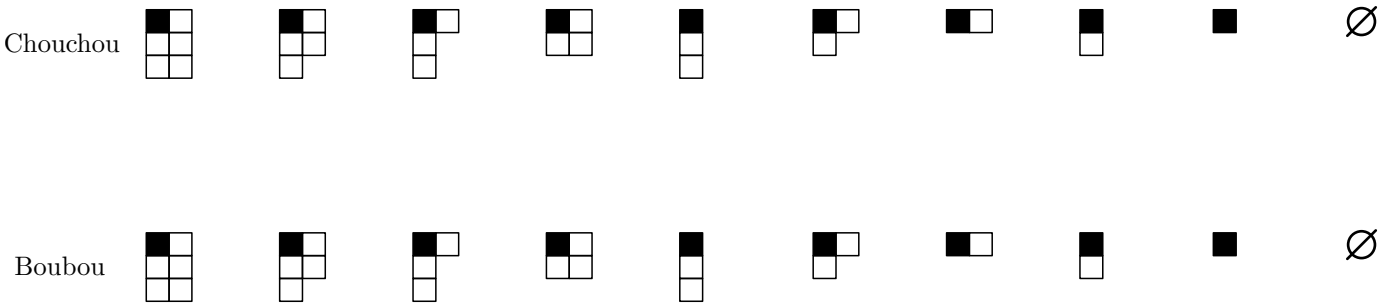
Remarque

Il s'agit d'une variante du "jeu des allumettes" présenté dans le cours. On regroupe ces jeux de stratégie basés sur des tas d'objets dans une catégorie appelée "jeux de Nim".

Existe-t-il une stratégie gagnante pour l'un des joueurs ?

Pour répondre à cette question, on va implémenter la méthode des attracteurs, étudiées en cours.

On représente ci-dessous les nœuds du graphe bipartis du jeu. La ligne du haut correspond aux situations aux Chouchou doit jouer, celle du bas représente les nœuds de Boubou.



1. **Compréhension sur un exemple.**

   $\mathbf{A_0}$: Identifier le nœud de Chouchou représentant sa victoire (position gagnante en zéro coup).

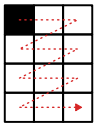
   $\mathbf{A_1}$: Faire apparaître les sommets de Boubou allant nécessairement vers cette position (préciser par des flèches les connexions).

   $\mathbf{A_2}$: Identifier les nœuds de Chouchou permettant d'aller vers au moins une position précédemment mise en valeur (ajouter les flèches entre les sommets pour justifier).

$\mathbf{A_k}$: Continuer à itérer ce processus afin de mettre en évidence l'attracteur final. Si Chouchou commence, peut-il gagner à coup sûr ?

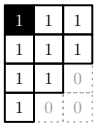
Implémentation en Python.

Pour représenter le graphe, on associera chaque nœud à une chaîne de caractère (appelée **identifiant**) contenant uniquement des `0` et des `1` :

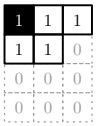


Sens lecture

Exemples:



111111110100



111110000000

On pourra ensuite facilement passer d'une chaîne de caractère à une matrice (`n`×`m`) (`n` = nombre de lignes, `m` = nombre de colonnes) et réciproquement avec les fonctions suivantes :

```
1  def string_to_array(string):
2      ''' Convertit une chaîne de '0' et '1' en matrice d'entiers de taille (n, m)
       '''
3      return np.array(list(string)).astype(int).reshape((n ,m))
4
5  def array_to_string(array):
6      ''' Convertit une matrice d'entiers en une chaîne de '0' et '1' '''
7      return ''.join(array.flatten().astype(str))
```

Bien sûr il faudra définir `n` et `m` en dehors des fonctions et importer `numpy` avec l'alias `np`.

2. Recopier ces fonctions et vérifier qu'elles fonctionnent correctement sur des exemples que vous choisirez.

Étant donné une taille de tablette `(n, m)`, on cherche à construire le graphe orienté reliant chaque position aux autres atteignables en un coup. On présentera cela sous la forme d'un dictionnaire dont :

- chaque clé sera l'identifiant d'un nœud ;
- la valeur associée à une clé donnée est une liste contenant tous les identifiants des nœuds accessibles en un coup.

3. Dans le cas où `n = m = 2`, écrire ci-dessous la définition du dictionnaire représentant le graphe du jeu :

```
graphe = {




}
```

4. Pour créer le graphe à partir de valeurs de `n` et `m`, on va diviser notre programme en trois grandes fonctions :
   - `prochain_noeud(position, i, j)`
     renvoie le prochain nœud si on part la position donnée et que le joueur choisit la case `(i, j)` ;
   - `noeuds_accessibles(id_position)`
     renvoie les identifiants des nœuds accessibles depuis la position donnée sous forme d'identifiant ;

› `creer_graphe(n, m)`
renvoie le dictionnaire représentant le graphe du jeu pour une tablette à `n` lignes et `m` colonnes.

a) Recopier et compléter la fonction `prochain_noeud` :

```
def prochain_noeud(position, i, j):

    # position et destination sont des matrices
    destination = position.copy()
    # ...

    return destination
```

b) Recopier et compléter la fonction `noeuds_accessibles` :

```
def noeuds_accessibles(id_position):

    # id_position est une chaîne de caractère
    position = string_to_array(id_position) # position est une matrice
    destinations = []

    for i in range(n):
        for j in range(m):
            # Vérifier si on peut enlever le carré (i, j)...
            # ... si oui, ajouter le noeud correspondant à la listes des destinations

    return destinations
```

c) Recopier et compléter la fonction `creer_graphe` :

```
def creer_graphe(n, m):

    tablette = np.ones((n, m)).astype(int)  # Créer la situation initiale: matrice remplie de 1
    noeuds_a_relier = [array_to_string(tablette)]   # Liste de noeuds explorés pour lesquels on n'a pas encore calculé les destinations
    graphe = {}

    while len(noeuds_a_relier) > 0:
        noeud = noeuds_a_relier[0]
        # Vérifier que l'on n'a pas déjà étudié ce noeud...
        # ... si ce n'est pas le cas, compléter graphe...
        # ... et ajouter à noeuds_a_relier les nouveaux noeuds apparus
        noeuds_a_relier.remove(noeud)   # Retire le noeud de la liste

    return graphe
```