TP - Algorithme des k plus proches voisins

> Abrégé *kppv* en français. En anglais, on dit *k nearest neighbors* souvent abrégé *knn*.

L'**algorithme des k plus proches voisins** appartient à la famille des algorithmes d'*apprentissage automatique* (machine learning) qui constituent le poumon de l'intelligence artificielle actuellement.

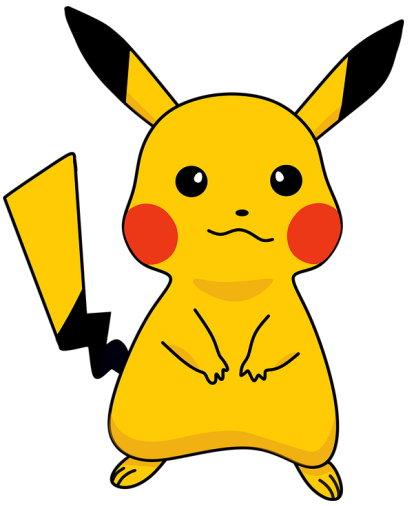
Pour simplifier, l'apprentissage automatique part souvent de données (data) et essaye de dire quelque chose des données qui n'ont pas encore été vues : il s'agit de *généraliser*, de *prédire*.

On va utiliser l'algorithme des k plus proches voisins pour résoudre un *problème de classification* : prédire la classe d'une donnée *inconnue* à partir de la classe des données connues.

Avant de décrire cet algorithme, introduisons la situation et le problème.

Présentation d'un problème de classification

Le jeu de données sur les Pokémons



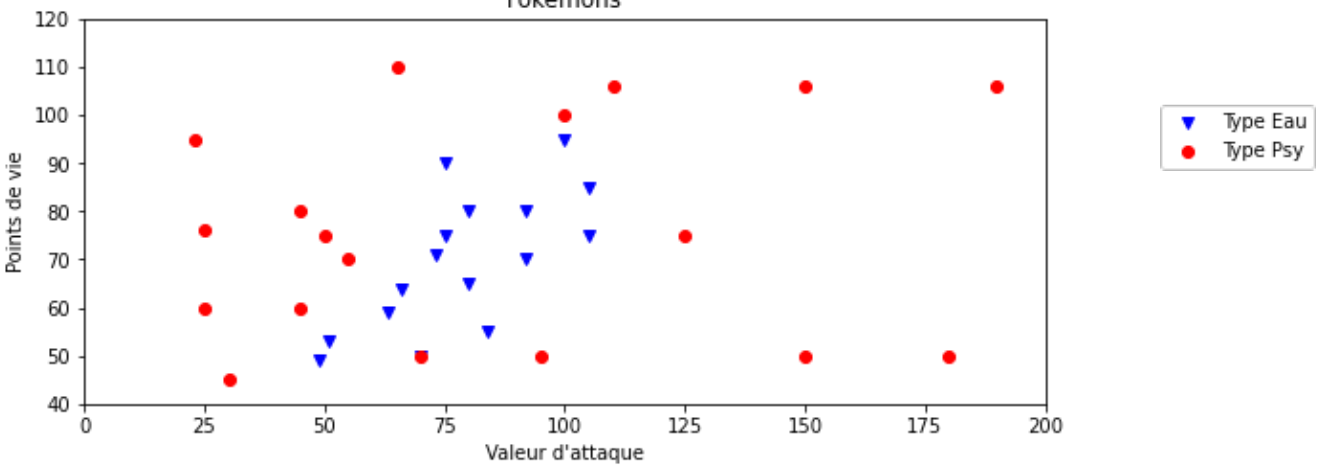
On dispose de données sur 34 Pokémons : leur type (Psy ou Eau), leur points de vie et la valeur de leurs attaques.

Ces données sont stockées dans le fichier CSV `pokemons.csv` que l'on s'empresse d'importer dans un tableau de dictionnaires appelé `pokemons` :

Entrée[ ]:
```
import csv
fichier = open('pokemons.csv', 'r', encoding = 'UTF-8')
t = csv.DictReader(fichier, delimiter=';')
pokemons = [dict(ligne) for ligne in t]  # création et construction du table
fichier.close()

pokemons  # pour voir les données importées
```

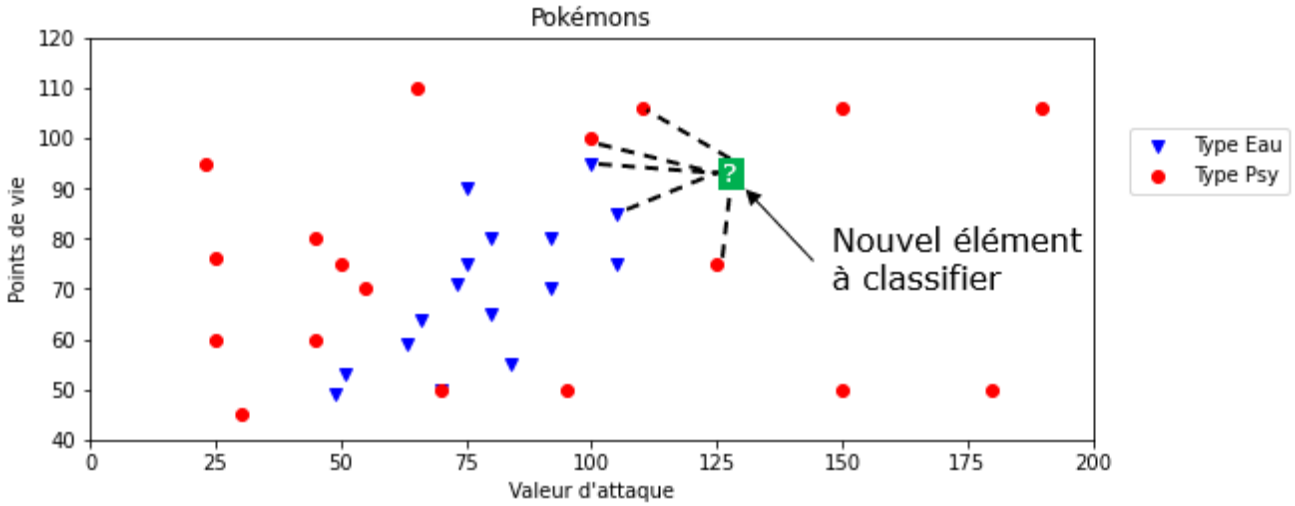
On peut représenter ces données graphiquement dans un repère orthonormé, avec les valeurs d'attaque en abscisses et les points de vie en ordonnées. Les Pokémons de type "Eau" sont représentés pas les triangles bleus, et ceux de type "Psy" par les points rouges.



C'est ce que vous allez faire par la suite !

Problème : peut-on prédire le type d'un nouveau Pokémon inconnu ?

On considère maintenant que l'on possède des données sur un nouveau Pokémon inconnu et on aimerait prédire s'il s'agit d'un Pokémon de type "Eau" ou de type "Psy".



Ce problème, qui demande à prédire à quelle catégorie, ou *classe*, appartient ce nouvel élément donné, est appelé *problème de classification*. L'algorithme des k plus proches voisins permet de trouver les k voisins les plus proches de ce nouvel élément dans le but de lui associer une *classe* plausible (Psy ou Eau, dans cet exemple). Par exemple, si k = 5 va chercher les 5 voisins les plus proches.

Algorithme des kppv

Énoncé et spécification

À partir d'un jeu de données `donnees` (par exemple, les données sur nos 34 Pokémons) et d'une donnée cible `cible` (le nouveau Pokemon à classifier), l'algorithme de k plus proches voisins doit déterminer les k données les plus proches de la cible.

Plus précisément, la spécification de l'algorithme des kppv est la suivante :

- **Entrées** :
  - une table `donnees` de taille n contenant les données et leurs classes

    - une donnée cible : `cible`
    - un nombre `k`
    - une règle permettant de calculer la *distance* entre deux données
- **Sortie** : une table `k_plus_proches_voisins`
- **Rôle** : trouver les `k` données les plus proches de `cible` parmi celles de la table `donnees` ("plus proches" au sens de la distance définie au départ)
- **Précondition** : $n \geq 1$ et $k <= n$.
- **Postcondition** : `k_plus_proches_voisins` contient les k plus proches voisins de `cible` parmi les données de la table `donnees`

Algorithme

L'algorithme *naif* des kppv s'exprime de manière simple :

1. Créer une table `distances_voisins` contenant les éléments de la table `donnees` et leurs distances avec la donnée `cible` .
2. Trier les données de la table `distances_voisins` selon la distance croissante avec la donnée `cible`
3. Renvoyer les `k` premiers éléments de cette table triée (`k_plus_proches_voisins`)

Implémentation de l'algorithme

L'algorithme des plus proches voisins repose presque entièrement sur la *distance* entre deux données. Il faut donc commencer par définir une distance entre deux données.

Étape 0 : Choix et implémentation de la distance

Dans la suite, on va choisir la distance "naturelle", c'est-à-dire celle "à vol d'oiseau". On parle de *distance euclidienne*.

Dans un repère orthonormé, si A et B ont pour coordonnées respectives (x_A, y_A) et (x_B, y_B) alors la distance (euclidienne) entre ces deux points est donnée par la formule :

$$\text{distance}(A, B) = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}.$$

> **Important**: Sachez cependant qu'il existe d'autres distances possibles et que le choix de la distance n'est pas anodin car ce choix peut aboutir à trouver des voisins les plus proches différents et donc conduire à des prédictions différentes (voir le résumé de cours !).

Dans notre cas, chaque Pokémon possède une abscisse égale à sa valeur d'attaque et une ordonnée égale à ses points de vie. Ainsi, la formule de la distance entre deux pokémons `d1` et `d2` se traduit ainsi :

$$\text{distance}(d1, d2) = \sqrt{(\text{valeur d'attaque de d2} - \text{valeur d'attaque de d1})^2 + (\text{points de vie de d2} -$$

✍️ **Question 1** : Calculez à la main la distance entre les deux Pokémons suivants.

```
p1 = {'Nom': 'Aligatueur', 'Type': 'Eau', 'Points de vie': '85', 'Attaqu
e': '105'}
p2 = {'Nom': 'Bargantua', 'Type': 'Eau', 'Points de vie': '70', 'Attaqu
e': '92'}
```

💻 **Question 2** : Écrivez les instructions permettant d'accéder :

- à la valeur d'attaque de `p1`
- au points de vie de `p2`

Entrée[ ]:
```
p1 = {'Nom': 'Aligatueur', 'Type': 'Eau', 'Points de vie': '85', 'Attaque':
p2 = {'Nom': 'inconnu', 'Type': 'inconnu', 'Points de vie': '92', 'Attaque':
```

Entrée[ ]:
```
# à vous de jouer !

```

Entrée[ ]:
```
# à vous de jouer !

```

💻 **Question 3** : Complétez la fonction `distance_euclidienne(d1, d2)` suivante qui renvoie la distance euclidienne entre deux Pokémons `d1` et `d2` .

> ⚠️ Il faut bien veiller à convertir les valeurs d'attaque et les points de vie en des nombres pour faire les calculs !*

Entrée[ ]:
```
# à vous de jouer !
from math import sqrt  # pour utiliser la racine carrée

def distance_euclidienne(d1, d2):
    x1 = ...  # abscisse de la donnée 1
    y1 = ...  # ordonnée de la donnée 1
    x2 = ...  # abscisse de la donnée 2
    y2 = ...  # ordonnée de la donnée 2
    return sqrt(...)  # formule donnant la distance euclidienne
```

💻 **Question 4** : Appelez cette fonction pour vérifier votre réponse à la question 1.

Entrée[ ]:
```
# à vous de jouer !

```

Étape 1 : Création de la table avec les distances

Maintenant que notre distance est définie, on peut passer à l'implémentation de l'algorithme. On va donc commencer par créer la table `distances_voisins` contenant les distances entre toutes nos données et la donnée `cible` .

On choisit de reprendre les dictionnaires de la table `pokemons` en leur ajoutant une clé correspondant à la distance avec la cible. Par exemple, le premier élément de `distances_voisins` sera un dictionnaire de la forme

```
{'Nom': 'Aligatueur', 'Type': 'Eau', 'Points de vie': '85', 'Attaque':
'105', 'distance': dist}
```

dans lequel `dist` est la distance à calculer entre le Pokémon `'Aligatueur'` et la cible.

💻 **Question 5** : Écrivez une fonction `table_avec_distances(donnees, cible)` qui renvoie la table `distances_voisins` contenant les éléments de la table `donnees` auxquels on a ajouté la clé `distance` dont la valeur est leur distance avec la donnée `cible` .

Exemple: Si `cible = {'Nom': 'inconnu', 'Type': 'inconnu', 'Points de vie': '92', 'Attaque': '127'}` alors l'appel `table_avec_distances(pokemons, cible)` renvoie le tableau dont le début est le suivant :

```
[{'Nom': 'Aligatueur',
  'Type': 'Eau',
  'Points de vie': '85',
  'Attaque': '105',
  'distance': 23.08679276123039},
 {'Nom': 'Bargantua',
  'Type': 'Eau',
  'Points de vie': '70',
  'Attaque': '92',
  'distance': 41.340053217188775},
 ...
]
```

Entrée[ ]:

```
def table_avec_distances(donnees, cible):
    # à vous de jouer !



# ESSAI
cible = {'Nom': 'inconnu', 'Type': 'inconnu', 'Points de vie': '92', 'Attaqu
distances_voisins = table_avec_distances(pokemons, cible)
distances_voisins
```

Étape 2 : Tri de la table selon la distance croissante

On va maintenant trier notre table de la plus petite à la plus grande distance avec notre cible, soit du plus proche au plus éloigné des voisins. Pour cela, on va utiliser la fonction `sorted` dont on peut afficher l'aide pour rappel :

Entrée[ ]:

```
help(sorted)
```

> Vous pouvez aussi revoir le TP sur les tris de table si nécessaire : TP - Trier une table de données (https://notebook.basthon.fr/?from=https://raw.githubusercontent.com/germainbecker/NSI/master/Premiere/Theme3_traitement_donnees_tables/TriTable.ipynb&aux=https://raw.githubusercontent.com/germainbecker/NSI/master/Premiere/Theme3_traitement_donnees_tables/eleves.csv&aux=https://raw.githubusercontent.com/germainbecker/NSI/master/Premiere/Theme3_traitement_donnees_tables/prenoms2021.csv#Trier-des-donn%C3%A9es-en-fonction-d'une-cl%C3%A9).

💻 **Question 6** : Créer une fonction `tri_distance(d)` qui servira de clé à notre tri.

Entrée[ ]:
```
# à vous de jouer !

```

💻 **Question 7** : Utilisez maintenant la fonction `sorted` pour créer une *nouvelle table* `distances_voisins_triee` contenant les données de `distances_voisins` triés par ordre croissant de distance.

Entrée[ ]:
```
cible = {'Nom': 'inconnu', 'Type': 'inconnu', 'Points de vie': '92', 'Attaqu
distances_voisins = table_avec_distances(pokemons, cible)

# à vous de jouer !



```

Étape 3 : Détermination des k plus proches voisins

Le plus dur est fait, il ne reste plus qu'à construire la table `k_plus_proches_voisins` contenant les k plus proches voisins. Comme la table `distances_voisins_triee` contient les voisins du plus proches au plus éloigné, il suffit de conserver les k premiers éléments de cette table !

💻 **Question 8** : Écrivez une fonction `conserve_k_premiers(k, table)` qui renvoie une nouvelle table contenant les `k` premiers éléments de la table `table` (où `k` est un entier positif inférieur ou égale à `len(table)`).

Entrée[ ]:
```
# à vous de jouer !

```

💻 **Question 9** : Vérifiez qu'en appelant cette fonction avec k = 3 et `table = distances_voisins_triee` vous obtenez bien les 3 premiers éléments de `distances_voisins_triee` .

Entrée[ ]:
```
# à vous de jouer !

```

Bilan

On peut maintenant regrouper tout ce qui vient d'être fait pour chaque étape pour écrire une fonction `kppv(donnees, cible, k)` qui renvoie les `k` plus proches voisins de `cible` dans `donnees` .

💻 **Question 10** : Complétez la fonction `kppv(donnees, cible, k)` suivante qui renvoie les `k` plus proches voisins de `cible` dans `donnees` .

Entrée[ ]:
```
# RECOPIE DES FONCTIONS UTILES :

from math import sqrt
# Calcul de la distance
def distance_euclidienne(d1, d2):
    # recopiez le code écrit à la question 3
    pass

# Fonction qui calcule et ajoute la distance entre la cible et chacune des d
def table_avec_distances(donnees, cible):
    # recopiez le code écrit à la question 5
    pass

# Clé du tri
def tri_distance(d):
    # recopiez le code écrit à la question 6
    pass



# À COMPLÉTER :

def kppv(donnees, cible, k):
    # étape 1 : création de la table avec les distances
    distances_voisins = ...

    # étape 2 : tri par distance croissante
    distances_voisins_triee = ...

    # étape 3 : récupération de k plus proches voisins
    k_plus_proches_voisins = ...

    return ...

```

💻 **Question 11** : Appelez la fonction la fonction `kppv` pour connaître les 3 plus proches voisins de notre cible `{'Nom': 'inconnu', 'Type': 'inconnu', 'Points de vie': '92', 'Attaque': '127'}` .

Entrée[ ]:
```
cible = {'Nom': 'inconnu', 'Type': 'inconnu', 'Points de vie': '92', 'Attaqu
# à vous de jouer !

```

💻 **Question 12** : Même question pour les 5 plus proches voisins puis pour les 9 plus proches voisins.

Entrée[ ]:
```
# à vous de jouer !

```

Et notre prédiction alors ?

L'algorithmes des kppv en lui-même n'apporte pas la réponse à notre problème de classification puisqu'il ne fournit que les k plus proches voisins (et leurs classes) de notre donnée cible. Il reste donc une dernière étape pour prédire la classe de notre nouvel élément : pour cela, on choisit la *classe majoritaire* (la plus présente) dans les k plus proches voisins.

> On est contents si k est impair car il ne peut pas y avoir d'ex-aequo !

✍️ **Question 13** : Donnez la classe majoritaire parmi les plus proches voisins lorsque k = 3, 5 puis 9. Pour chaque valeur de k, quelle serait la prédiction pour notre Pokémon cible inconnu ?

```
Réponse :

```

> **Moralité** : La valeur du paramètre `k` est importante car elle a une influence forte sur la prédiction. En pratique, elle doit être judicieusement choisie : 👉 voir dans le cours comment trouver expérimentalement une bonne valeur de k.

Visualisation graphique

On peut utiliser le module `matplotlib` pour représenter graphiquement nos Pokémons, et observer graphiquement les plus proches voisins selon la valeur de `k` choisies.

Pour créer une nuage de points, il suffit d'utiliser la fonction `plot` qui prend en paramètres deux tableaux contenant respectivement la liste des abscisses et la liste des ordonnées des points à construire, ainsi qu'un paramètre permettant de définir le type de points et leur couleur :

Entrée[ ]:
```
import matplotlib.pyplot as plt

liste_abscisses = [0, 2, 5, 9, 7]  # tableau avec les abscisses
liste_ordonnees = [5, 7, 12, 3, 8]  # tableau avec les ordonnées

plt.plot(liste_abscisses, liste_ordonnees,'ro')  # r pour red, o pour un cer

plt.xlim(-2, 12)  # pour définir la fenêtre en abscisse
plt.ylim(0, 14)  # pour définir la fenêtre en ordonnée
plt.gca().set_aspect('equal', adjustable='box')  # pour que le repère soit o

plt.show()  # affichage du graphique
```

Pour représenter nos Pokémons sous forme de nuages de points (valeur d'attaque en abscisse et points de vie en ordonnée), il suffit alors de créer un tableau contenant les abscisses (valeur d'attaque) et un tableau contenant les ordonnées (points de vie) puis d'utiliser la fonction `plot` pour construire les points.

Pour bien visualiser les deux classes, on va créer un nuage de points pour chaque classe : les Pokémons de type "Eau" seront représentés par des points bleus et les Pokémons "Psy" par des points rouges. On crée aussi le point correspondant à notre Pokémon cible en cyan :

Entrée[ ]:

```
import matplotlib.pyplot as plt

# construction des Pokémons de type 'Eau'
valeur_attaque_eau = [int(p['Attaque']) for p in pokemons if p['Type'] == 'E
points_de_vie_eau = [int(p['Points de vie']) for p in pokemons if p['Type']
plt.plot(valeur_attaque_eau, points_de_vie_eau, 'bv')  # construction du gra

# construction des Pokémons de type 'Psy'
valeur_attaque_psy = [int(p['Attaque']) for p in pokemons if p['Type'] == 'P
points_de_vie_psy = [int(p['Points de vie']) for p in pokemons if p['Type']
plt.plot(valeur_attaque_psy, points_de_vie_psy, 'ro')  # construction du gra

# construction du Pokémon cible
cible = {'Nom': 'inconnu', 'Type': 'inconnu', 'Points de vie': '92', 'Attaqu
valeur_attaque_cible = int(cible['Attaque'])
points_de_vie_cible = int(cible['Points de vie'])
plt.plot(valeur_attaque_cible, points_de_vie_cible, 'cD')


# Pour définir le titre, noms de axes et la légende
plt.title("Pokémons")
plt.xlabel("Valeur d'attaque")
plt.ylabel("Points de vie")
plt.legend(["Type Eau", "Type Psy"], loc='upper center', bbox_to_anchor=(1.2

plt.xlim(0, 200)
plt.ylim(40, 120)
plt.gca().set_aspect('equal', adjustable='box')

plt.show()
```

💻 **Question 14** : Complétez ci-dessous le programme précédent pour construire également le nuage de points correspondant aux k plus plus proches voisins déterminés par notre algorithme. Vérifiez ensuite graphiquement pour différentes valeurs de k que les plus proches voisins sont les bons.

Entrée[ ]:

```
import matplotlib.pyplot as plt

# construction des Pokémons de type 'Eau'
valeur_attaque_eau = [int(p['Attaque']) for p in pokemons if p['Type'] == 'E
points_de_vie_eau = [int(p['Points de vie']) for p in pokemons if p['Type']
plt.plot(valeur_attaque_eau, points_de_vie_eau, 'bv')  # construction du gra

# construction des Pokémons de type 'Psy'
valeur_attaque_psy = [int(p['Attaque']) for p in pokemons if p['Type'] == 'P
points_de_vie_psy = [int(p['Points de vie']) for p in pokemons if p['Type']
plt.plot(valeur_attaque_psy, points_de_vie_psy, 'ro')  # construction du gra

# construction du Pokémon cible
cible = {'Nom': 'inconnu', 'Type': 'inconnu', 'Points de vie': '92', 'Attaqu
valeur_attaque_cible = int(cible['Attaque'])
points_de_vie_cible = int(cible['Points de vie'])
plt.plot(valeur_attaque_cible, points_de_vie_cible, 'cD')


# construction des KPPV
# À COMPLÉTER :









# Pour définir le titre, noms de axes et la légende
plt.title("Pokémons")
plt.xlabel("Valeur d'attaque")
plt.ylabel("Points de vie")
plt.legend(["Type Eau", "Type Psy"], loc='upper center', bbox_to_anchor=(1.2

plt.xlim(0, 200)
plt.ylim(40, 120)
plt.gca().set_aspect('equal', adjustable='box')

plt.show()
```

Et si on change de distance ?

On a utilisé la distance euclidienne pour mesurer la distance entre nos données et la cible. C'est la distance "naturelle" mais il existe d'autres distances. Par exemple, la **distance de Manhattan** entre deux données $d_1(x_1, y_1)$ et $d_2(x_2, y_2)$ est définie ainsi
$\text{distance\_manhattan}(d_1, d_2) = |x_2 - x_1| + |y_2 - y_1|$.

> Article Wikipédia : Distance de Manhattan (https://fr.wikipedia.org/wiki/Distance_de_Manhattan)

On peut modifier notre programme simplement en définissant et en utilisant cette nouvelle distance :

Entrée[ ]:
```python
# définition de la distance utilisée
def distance_manhattan(d1, d2):
    x1 = float(d1['Attaque'])
    y1 = float(d1['Points de vie'])
    x2 = float(d2['Attaque'])
    y2 = float(d2['Points de vie'])
    return abs(x2-x1) + abs(y2-y1)  # abs est la fonction valeur absolue

def table_avec_distances(donnees, cible):
    distances_voisins = [d for d in donnees]
    for d in distances_voisins:
        distance = distance_manhattan(d, cible)  # CHANGEMENT DE DISTANCE
        d['distance'] = distance
    return distances_voisins

def tri_distance(d):
    return d['distance']

def kppv(donnees, cible, k):
    # étape 1 : création de la table avec les distances
    distances_voisins = table_avec_distances(donnees, cible)
    # étape 2 : tri par distance croissante
    distances_voisins_triee = sorted(distances_voisins, key=tri_distance)
    # étape 3 : récupération de k plus proches voisins
    k_plus_proches_voisins = conserve_k_premiers(k, distances_voisins_triee)

    return k_plus_proches_voisins

```

En déterminant les 3 plus proches voisins on obtient :

Entrée[ ]:
```python
cible = {'Nom': 'inconnu', 'Type': 'inconnu', 'Points de vie': '92', 'Attaqu
kppv(pokemons, cible, 3)
```

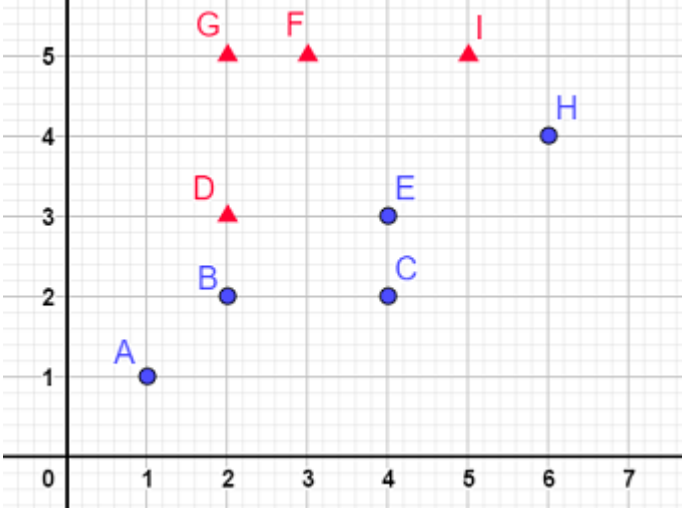
Dans ce cas, pour k = 3, on prédirait que notre Pokémon inconnu est de type "Eau" (2 "Eau" contre 1 "Psy") et la prédiction serait l'inverse de celle utilisant la distance euclidienne !

> **Moralité** : La distance utilisée a également toute son importance puisque son choix peut aboutir à des prédictions différentes. De plus, nous n'avons parlé ici que de distances *géométriques* s'appliquant à des données chiffrées. Toutes les données ne sont pas adaptées à ce type de distance : si on veut comparer la distance entre deux chaînes de caractères (dans le but de prédire la langue d'origine de certains mots par exemple) d'autres types de distances sont à considérer : *la distance de Hamming* ou la *distance d'édition* qui seront abordées en Terminale.

EXERCICES

Exercice 1

On donne des points dans un repère orthonormé. Chaque point possède une classe : Rouge ou Bleue. Dans cette exercice, on utilise la **distance euclidienne** (naturelle) pour l'algorithme des k plus proches voisins.



✍️ **Question 1** : On considère le point $M(1, 3)$. Appliquez l'algorithme des 3 plus proches voisins à ce point puis donnez une prédiction quant à sa classe.

✍️ **Question 2** : Même question avec le point $N(4, 4)$.

✍️ **Question 3** : Donnez une valeur de k qui donnerait la prédiction "Bleue" pour le point N après avoir appliqué l'algorithme des k plus proches voisins.

Exercice 2

On peut ainsi représenter l'ensemble des points de l'exercice 1 dans un tableau `donnees` comme ci-dessous :

Entrée[ ]:
```
A = {'nom': 'A', 'abs': 1, 'ord': 1, 'classe': 'Bleu'}
B = {'nom': 'B', 'abs': 2, 'ord': 2, 'classe': 'Bleu'}
C = {'nom': 'C', 'abs': 4, 'ord': 2, 'classe': 'Bleu'}
D = {'nom': 'D', 'abs': 2, 'ord': 3, 'classe': 'Rouge'}
E = {'nom': 'E', 'abs': 4, 'ord': 3, 'classe': 'Bleu'}
F = {'nom': 'F', 'abs': 3, 'ord': 5, 'classe': 'Rouge'}
G = {'nom': 'G', 'abs': 2, 'ord': 5, 'classe': 'Rouge'}
H = {'nom': 'H', 'abs': 6, 'ord': 4, 'classe': 'Bleu'}
I = {'nom': 'I', 'abs': 5, 'ord': 5, 'classe': 'Rouge'}

donnees = [A, B, C, D, E, F, G, H, I]
donnees
```

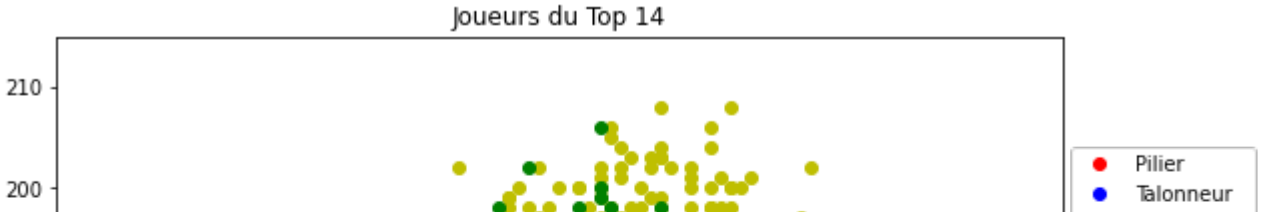
💻 **Question** : Utilisez la fonction `kppv` définie dans le TP sur l'algorithme des k plus proches voisins (question 10) pour vérifier vos réponses aux questions de la partie A. *Vous utiliserez la distance euclidienne qu'il faudra bien sûr légèrement adapter au format de nos données*.

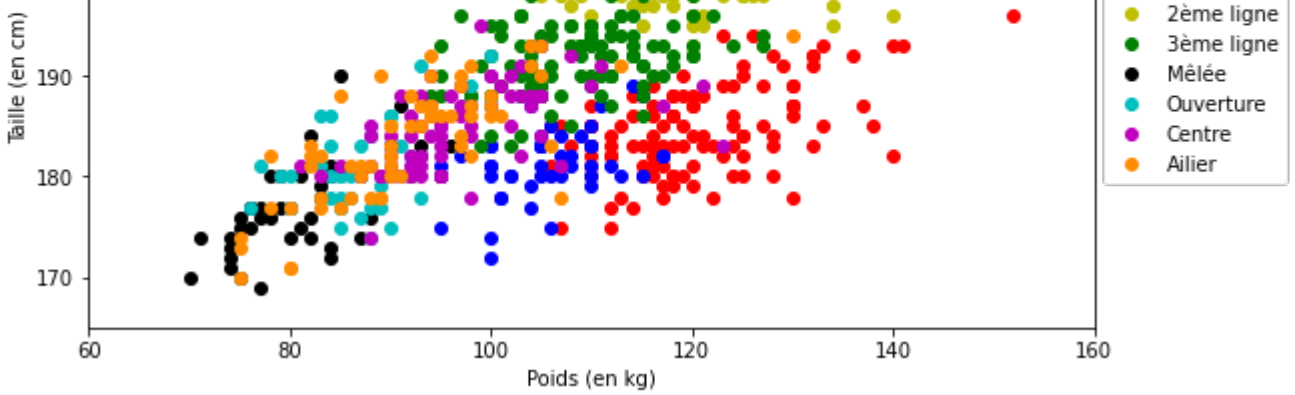
Entrée[ ]:
```
# à vous de jouer !

```

Exercice 3 : Top 14

Le fichier `top14.csv` contient des données sur tous les joueurs du top 14 de rugby. On peut représenter ces données de la façon suivante.





💻 **Question** : Vous rencontrez une personne souhaitant jouer au rugby. Elle vous donne sa taille (180 cm) et son poids (100 kg). Vous allez utiliser l'algorithme des k plus proches voisins pour lui indiquer quel poste est le plus proche de ses caractéristiques physiques. *Essayez plusieurs valeurs de k et n'hésitez pas à représenter graphiquement les données comme sur l'image au-dessus.*

Entrée[ ]:

```
1 # à vous de jouer !
2
```

Références :

- Équipe pédagogique du DIU EIL, Université de Nantes (C. JERMANN).
- Prepabac, spécialité NSI 1ère, C. ADOBET, G. CONNAN, G. ROZSAVOLGYI, L. SIGNAC (pour l'idée de l'activité).
- Cours de G. LASSUS pour l'idée de l'exercice 3 sur les joueurs du Top 14 : lien vers son Notebook sur Github (https://github.com/glassus/nsi/blob/master/Premiere/Theme05_Algorithmique/07_Algorithme_KNN.ipynb)

Germain BECKER & Sébastien POINT, Lycée Mounier, ANGERS