Algorithme des k plus proches voisins

Introduction

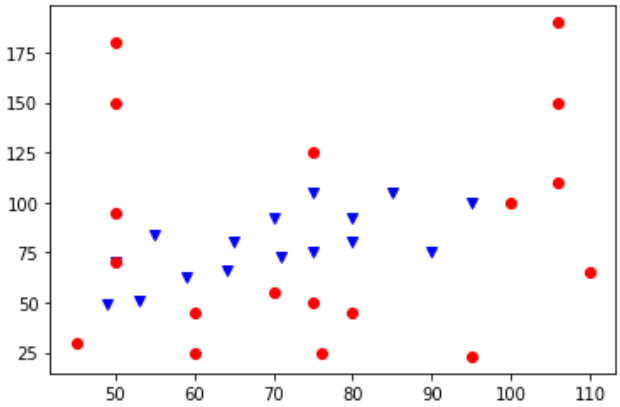
L'algorithme des k plus proches voisins est abrégé *kppv* en français. En anglais, on dit *k nearest neighbors* souvent abrégé *knn*.

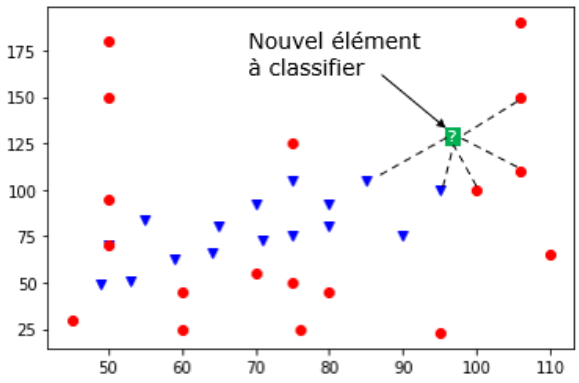
L'algorithme des k plus proches voisins appartient à la famille des algorithmes d'*apprentissage automatique* (machine learning) qui constituent le poumon de l'intelligence artificielle actuellement.

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*.

Exemple d'introduction

On dispose de données sur 34 Pokémons : leur type (Psy ou Eau), leur points de vie et la valeur de leurs attaques. On peut représenter ces données graphiquement, avec les points de vie en abscisses et les valeurs d'attaque en ordonnées.





Les Pokémons de type "Eau" sont représentés pas les points bleus, et ceux de type "Psy" par les points rouges.

Peut-on prédire le type d'un nouveau Pokemon inconnu ?

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 (si k = 5 on cherche les 5 voisins les plus proches) de ce nouvel élément dans le but de lui associer une *classe* plausible (Psy ou Eau, dans cet exemple).

Algorithme naïf des kppv

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

Voici un algorithme permettant de résoudre ce problème :

Données et préconditions :

- une table `donnees` de taille `n` contenant les données et leurs classes
- une donnée cible : `cible`
- un nombre `k` inférieur à `n`
- une règle permettant de calculer la *distance* entre deux données

Résultat : un tableau contenant les `k` plus proches voisins de la donnée cible.

Algorithme :

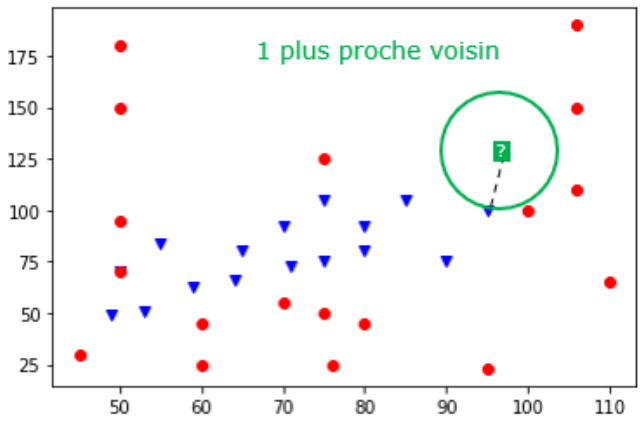
1. Créer une table `distances_voisins` contenant les éléments de la table `donnees` et leurs distances avec la donnée `cible`.
2. Trier les données de la table `distances_voisins` selon la distance croissante avec la donnée `cible`
3. Renvoyer les `k` premiers éléments de cette table triée.

Et notre prédiction alors ?

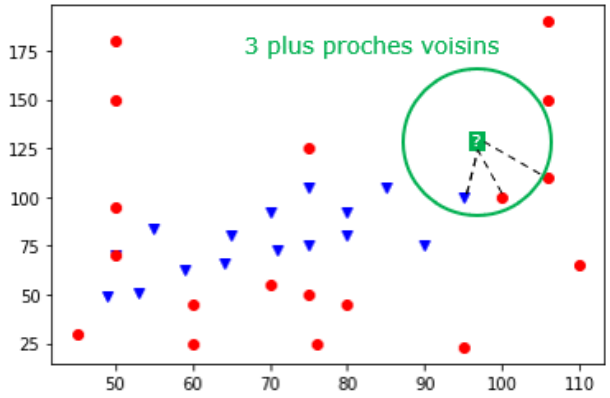
L'algorithmes des kppv en lui-même n'apporte pas la réponse à notre problème de classification puisqu'il ne fournit que les k plus proches voisins (et leurs classes) de notre donnée cible. Il reste donc une dernière étape pour prédire la classe de notre nouvel élément : pour cela, on choisit la *classe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

Influence de la valeur de k

La valeur de k est très importante, elle doit être choisie judicieusement car elle a une influence forte sur la prédiction. Regardons le résultat de la prédiction pour différentes valeurs de k sur notre exemple.



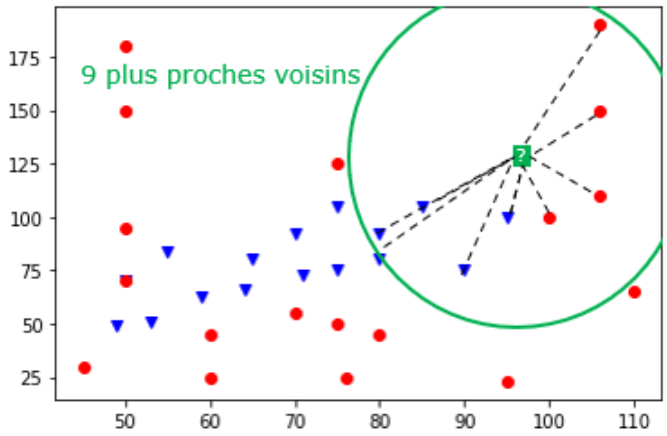
Exemple 1 : le 1 plus proche voisin
Si k=1k=1, cela revient à chercher la donnée la plus proche de notre élément cible. Ici, on se rend compte que sa classe est "Eau" (point bleu) donc on classerait le nouveau Pokémon comme étant de type "Eau".



Exemple 2 : les 3 plus proches voisins

On se rend compte que la classe majoritaire dans les 3 plus proches voisins est "Psy" (2 points rouges contre 1 point bleu) donc on classerait le Pokemon inconnu comme étant de type "Psy".

Exemple 3 : les 9 plus proches voisins



On se rend compte que la classe majoritaire dans les 9 plus proches voisins est "Eau" (5 points bleus contre 4 points rouges) donc on classerait le Pokemon inconnu comme étant de type "Eau".

En poursuivant, si on choisit k=34 (le nombre total de données), alors la prédiction serait toujours "Psy" car c'est la classe majoritaire de l'échantillon. Il est donc incorrect de pense que plus la valeur de kk augmente meilleure sera la prédiction, c'est plus complexe que cela.
C'est bien beau tout ça, mais quelle valeur de k faut-il choisir ?

Choix de la valeur de k par expérimentation

Pour trouver une bonne valeur de k il est possible d'appliquer le protocole expérimental suivant :

- Séparer les données en deux paquets : un paquet pour *entraîner le modèle* (90 % par exemple), un second pour *tester le modèle* (les 10% restants)
- Utiliser le premier paquet comme nouveau jeu de données et appliquer l'algorithme des kppv sur les éléments qui ont été retirés
- Comparer les réponses de l'algorithme avec les réponses attendues (on connaît la classe des éléments retirés donc on peut comparer)

En appliquant ce protocole à différentes valeurs de k, on peut déterminer quelle valeur fournit les meilleurs résultats. Il est même judicieux de recommencer en retirant d'autres données pour affiner encore davantage la recherche de la meilleure valeur k : on parle alors de *validation croisée* qui est une méthode d'apprentissage.

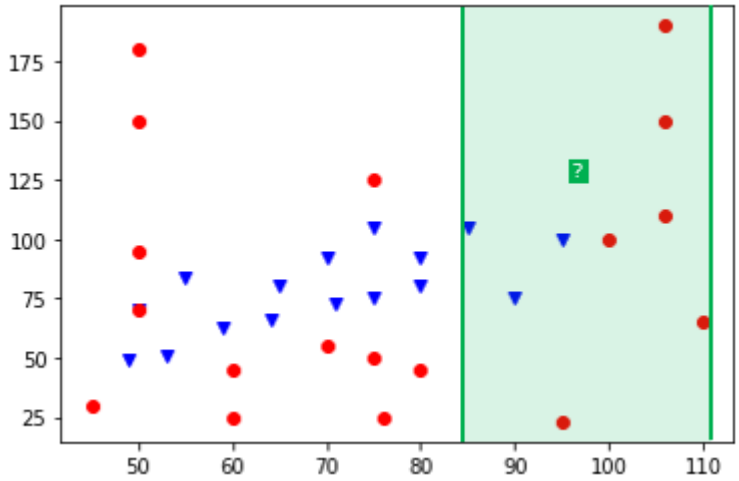
Choix de la distance

L'algorithme des plus proches voisins repose presque entièrement sur la *distance* entre deux données. Dans les exemples vus précédemment, c'est la distance "naturelle" qui a été choisie (celle "à vol d'oiseau").
Dans un repère orthonormé, si AA et BB ont pour coordonnées respectives (x_A,y_A)(xA,yA) et (x_B,y_B)(xB,yB) alors la distance entre ces deux points est donnée par la formule :

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

On parle alors de la *distance euclidienne*. Sachez cependant qu'il existe d'autres distances et vous en rencontrerez dans les exercices. Par exemple, on peut très bien imaginer que les valeurs sur l'axe des ordonnées ne nous intéressent pas et utiliser une distance ne prenant en compte que l'axe des abscisses avec la formule : $\text{distance(A,B)}=|x_B-x_A|$. Ainsi, sur notre exemple, et avec cette distance, la classe majoritaire des 9 plus proches voisins de notre nouveau Pokémon est "Psy" (6 points rouges contre 3 points bleus), ce qui donnerait une prédiction contraire à celle donnée en utilisant la distance "naturelle" (euclidienne).

Moralité : On voit donc que le choix de la distance n'est pas anodin et que ce choix peut aboutir à des prédictions différentes.



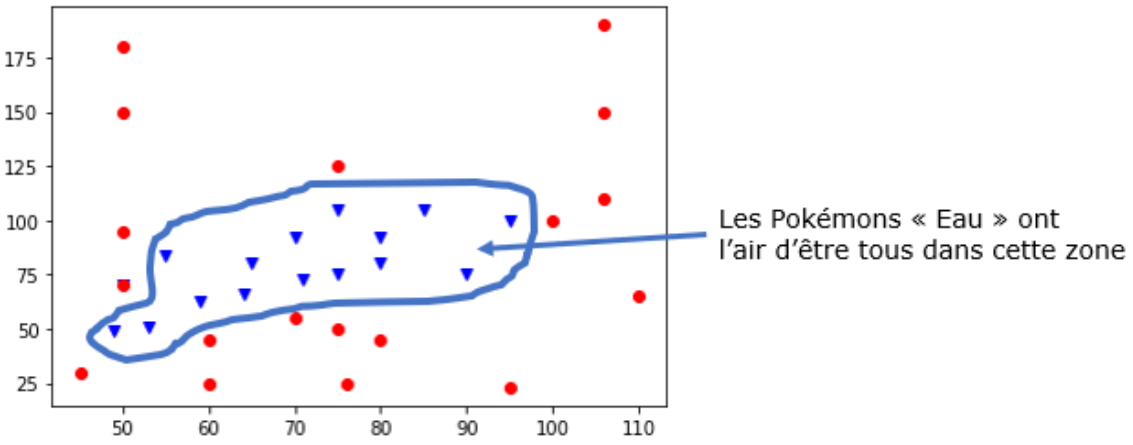
Remarque : Nous n'avons parlé ici que de distances *géométriques* qui ne s'appliquent qu'à des données chiffrées. Toutes les données ne sont pas adaptées à ce type de distance : si on veut comparer la distance entre deux chaînes de caractères (dans le but de prédire la langue d'origine de certains mots par exemple) d'autres types de distances sont à considérer : *la distance de Hamming* ou la *distance d'édition* qui seront abordées en Terminale.

Un algorithme d'*apprentissage* ? d'*IA* ?

Apprentissage par coeur

La plupart des algorithmes d'apprentissage automatique cherchent à *apprendre* quelque chose du jeu de données qui lui est fourni, c'est-à-dire à remplacer les données par un *modèle* (une sorte de "règle" permettant de classer les données, de prendre une décision, etc.). Autrement dit, un tel algorithme tente de "comprendre" les données pour en déduire un modèle, on peut voir cela comme un apprentissage *intelligent*.

Pour illustrer cela de manière simplifiée, un algorithme d'apprentissage automatique *intelligent* chercherait par exemple, à partir des données, à délimiter deux "zones" séparant les Pokémons "Psy" des Pokémons "Eau". Il n'aurait alors plus besoin des données (des points) pour prédire la classe d'un nouvel élément car il suffirait de regarder dans quelle zone celui-ci se trouve (d'utiliser le modèle qu'il a deviné).



L'algorithme des k plus proches voisins n'a pas cette "intelligence" car il n'essaie pas de construire un modèle mais se sert des données elles-mêmes pour donner un résultat : on parle alors d'**apprentissage par coeur**.

Intelligence Artificielle

L'algorithme des k plus proches voisins est-il alors un algorithme d'intelligence artificielle ?

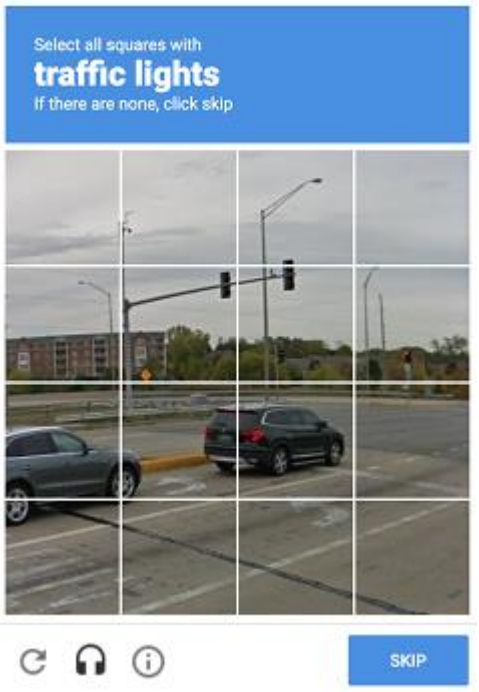
L'apprentissage par coeur est clairement l'apprentissage le moins intelligent mais la réponse est OUI si on considère sa capacité de prédiction : classifier un élément qu'il n'a jamais vu.

Les données et l'apprentissage automatique

Big data et deep learning

Le **deep learning** (ou *apprentissage profond*) est une famille de méthode d'apprentissage automatique. En 2012, les techniques de deep learning ont réussi à fonctionner et obtenir des résultats spectaculaires. Ces résultats ont été rendus possibles notamment par l'arrivée de données en volume massifs (**big data**) permettant aux algorithmes d'apprendre à résoudre un problème. Ces données en masse permettent de disposer d'une importante quantité de données sur lesquelles entrainer et affiner les algorithmes.

Les stratégies mises en place par les géants du numérique (GAFAM) tournent entièrement autour de la récolte de données de leurs clients sur n'importe quel sujet pour "nourrir" leurs algorithmes d'apprentissage. C'est ainsi qu'Amazon arrive à nous proposer des "suggestions d'achat". Google quant à lui nous utilise lorsque nous devons prouver que nous ne sommes pas un robot. En effet, qui n'a pas déjà vu un écran de ce genre ?



En cliquant aux bons endroits, on assigne une classe aux différentes parties de l'image (feu de circulation ou non) et celles-ci viennent alimenter les bases de données sur lesquelles les algorithmes s'entraînent. Ils serviront ensuite aux voitures autonomes de Google à repérer les feux tricolores sur la route.

Qualité des données et dangers

La qualité des données est primordiale dans l'apprentissage automatique car ce sont elles qui définissent presque entièrement la qualité des résultats des algorithmes. Voici quelques exemples, en vrac :

- Si la base de données utilisée pour entraîner un algorithme de détourage n'est constituée que de photos d'humains, il y a des fortes chances que l'algorithme ne parvienne pas à détourer un chat correctement. C'était le cas au départ pour l'algorithme du site remove.bg mais les données d'entraînement sont désormais plus complètes et les détourages fournissent des résultats impressionnants.

- L’algorithme mis en place à partir de 2015 par Amazon (source) pour faciliter le recrutement de talents utilisait des données de centaines de milliers de curriculum vitae (CV) reçus par Amazon au cours des dix dernières années en vue de noter de nouvelles candidatures. L’algorithme a été rapidement suspendu car il discriminait grandement les femmes. En effet, les CV d'entraînement comprenait une écrasante majorité d'hommes, l’algorithme ne laissant du coup aucune chance aux nouvelles candidates pourtant qualifiées. On dit dans ce cas que les données sont *biaisées*.
- Aux Etats-unis, ils prédisent les taux de criminalité dans les quartiers et déploient les effectifs policiers en conséquence. Mais les données sur lesquelles les systèmes sont entrainés sont également biaisés car déséquilibrés avec davantage de personnes de couleurs par exemple (voir source).
- ...

“*Your system is only as good as the data that you use to train it on*”, Kate Crawford, cofounder and co-director of AI Now

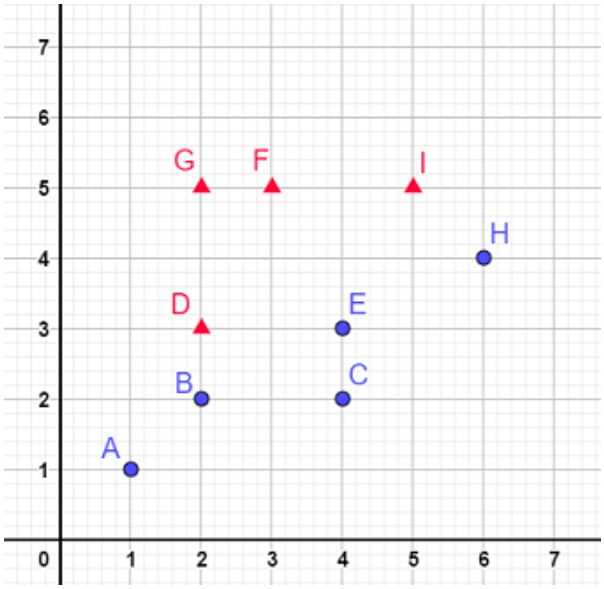
Conclusion

- Nous avons vu que l'algorithme des kk plus proches voisins faisait partie de la famille des algorithmes d'apprentissage automatique (*machnine learning* en anglais) qui se nourissent de données pour prédire des choses sur une donnée inconnue. En particulier, il permet de résoudre des problèmes de classification.
- L'algorithme de kppv permet de trouver les kk voisins les plus proches d'une nouvelle donnée. Il est donc nécessaire de lui associer une distance pour apprécier cette notion de proximité.
- On peut alors prédire la classe d'une nouvelle donnée en prenant celle qui est majoritaire parmi ses plus proches voisins.
- Les prédictions varient selon la valeur de kk et selon la distance choisie : en pratique il est donc important de bien choisir ce deux données.
- Les algorithmes d'apprentissage automatique (IA) se sont beaucoup développés depuis les années 2010 grâce au *big data* qui leur permet de s'entraîner sur un très grand nombre de données.
- La qualité des résultats dépend grandement de la qualité des données d'entraînement : il est donc important de s'assurer de leur qualité pour éviter des *biais* pouvant être discriminants, racistes...

Avec Python

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.



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.
2. Même question avec le point $N(4,4)$.
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 : implémentation de plusieurs distances

L'objectif de cet exercice est d'écrire trois distances possibles afin de pouvoir programmer implémenter l'algorithme de kk plus proches voisin dans l'exercice suivant. Les trois distances choisies sont : la distance euclidienne, la distance de Manhattan et la distance de Tchebychev dont les formules sont données ci-dessous.

On considère deux données d_1 et d_2 de coordonnées respectives (x_1,y_1) et (x_2,y_2) dans un repère orthonormé. Voici les formules donnant les distances dans chaque cas :

- Distance **euclidienne** : $\text{distance}(d_1,d_2)=\sqrt{(x_B-x_A)^2+(y_B-y_A)^2}$.
- Distance **de Manhattan** : $\text{distance}(d_1,d_2)=|x_2-x_1|+|y_2-y_1|$.
- Distance **de Tchebychev** : $\text{distance}(d_1,d_2)=\max(|x_2-x_1|,|y_2-y_1|)$.

Représentation des données

On considère ici que les données sont représentées par des dictionnaires. Par exemple, le point $D(2,3)$ de l'exercice précédent est représenté par le dictionnaire :

```
{'nom': 'D', 'abs': 2, 'ord': 3, 'classe': 'Rouge'}
```

On peut ainsi représenter l'ensemble des points de l'exercice 1.

On trouvera ces points dans le repl : https://replit.com/@jkergot/k-plus-proches-voisins#main.py

Distance euclidienne

La fonction `distance_euclidienne(d1, d2)` suivante renvoie la distance euclidienne entre les données `d1` et `d2` (dictionnaires ayant des clés représentant une abscisse et une ordonnée).

```
def distance_euclidienne(d1, d2):
```

On trouvera la fonction dans le repl

Voici quelques assertions pour vérifier le fonctionnement de la fonction

```
assert distance_euclidienne(B, D) == 1  # la distance euclidienne entre les points (2,2) et (2,3) (B et D)
assert distance_euclidienne(A, B) == sqrt(2)          assert distance_euclidienne(D, D) == 0
assert distance_euclidienne(B, F) == sqrt(10)
```

Distance de Manhattan

Question 1 Ecrivez dans votre fork du repl une fonction `distance_manhattan(d1, d2)` qui renvoie la distance de Manhattan entre les données `d1` et `d2`. On rappelle que la fonction valeur absolue est notée `abs` en Python. Quelques assertions devant être vérifiées par la fonction sont données ci-dessous.

```
# à compléter
```

Quelques assertions pour vérifier votre travail

```
assert distance_manhattan(A, B) == 2
assert distance_manhattan(E, F) == 3
assert distance_manhattan(D, H) == 5
assert distance_manhattan(D, D) == 0
```

Distance de Tchebychev

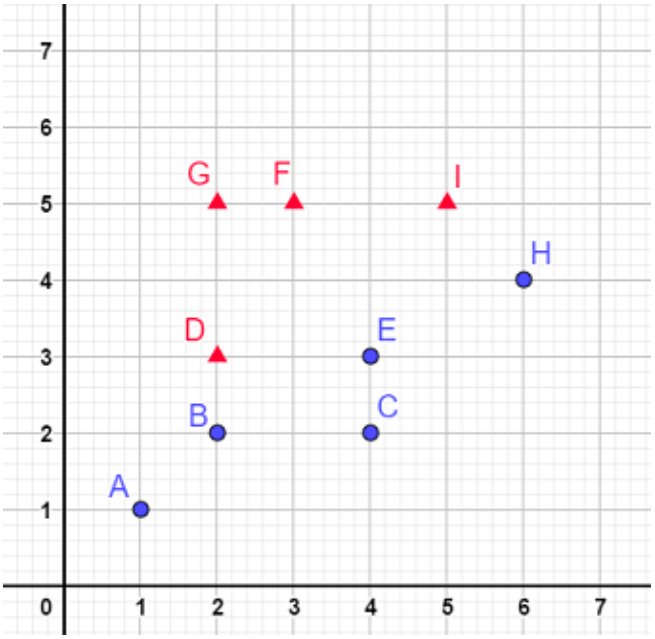
Question 2 : Ecrivez une fonction `distance_tchebychev(m1, m2)` qui renvoie la distance de Tchebychev entre les données `d1` et `d2`. On rappelle que la fonction valeur absolue est notée `abs` en Python. Vous commencerez par écrire quelques assertions devant être vérifiées par la fonction.

```
# à compléter
```

```
# tests à écrire (assertions) :
```

Exercice 3 : implémentation de l'algo des kppv

On va travailler avec les données de l'exercice 1 dont on redonne la figure.



Représentation des données

Comme dans l'exercice 2, chaque point est représenté par un dictionnaire et on peut mémoriser dans une table `donnees` l'ensemble des points de la figure.

```
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]
```

Algorithme des kppv avec la distance euclidienne

On va maintenant implémenter l'algorithme des kk plus proches voisins (abrégé *kppv*) que l'on rappelle :

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.

Première étape : création de la table avec les distances

Il faut commencer par choisir comment représenter les éléments de la table `distances_voisins`. Ici, il a été décidé de reprendre les dictionnaires de la table `donnees` en leur **ajoutant une clé correspondant à la distance avec la cible**.

Par exemple, le premier élément de `distances_voisins` sera le dictionnaire

```
{'nom': 'A', 'abs': 1, 'ord': 1, 'classe': 'Bleu', 'distance' : dist}
```

qui est le premier dictionnaire des données auquel on a ajouté une clé `'distance'` dont la valeur `dist` est la distance à calculer entre le point `'A'` et la cible.
On aurait aussi pu choisir d'utiliser un tableau ou même de ne conserver que les distances en travaillant par la suite qu'avec les indices pour retrouver les plus proches voisins.

Question 1 : Construire la table `distances_voisins` à partir de la table `donnees` en prenant le point $M(1,3)$M(1,3) comme `cible` et la distance euclidienne comme distance.

```
cible = {'nom': 'M', 'abs': 1, 'ord': 3}
# à vous de jouer !
```

Question 2 :
Comparez la table `distances_voisins` obtenue avec les réponses obtenues dans l'exercice 1.

```
# à vous de jouer !
```

Deuxième étape : tri de la table

Il faut désormais trier la table obtenue par ordre croissant des distances.

Question 3 : Utilisez la fonction `sorted` pour créer une nouvelle table `distances_voisins_triee` qui contient les éléments de la table `distances_voisins` triés par ordre croissant selon la clé `'distance'`. *// faudra au préalable définir la fonction de tri.*
Se référer si besoin au paragraphe "Trier des données en fonction d'une clé" du TP : Tri d'une table

Question 4 : Vérifiez que les données ont bien été triées par ordre croissant des distances avec le point $M(1,3)$M(1,3) : les points DD, BB, AA, GG, FF, EE, CC, II et HH dans cet ordre.

Troisième étape : récupérer les kk plus proches voisins

Pour récupérer les kk plus proches voisins il suffit de récupérer les kk premiers éléments de la table `distances_voisins_triee`. Avec une boucle `for` cela donnerait

```
k = 3
k_plus_proches_voisins = [distances_voisins_triee[i] for i in range(k)]
k_plus_proches_voisins
```

On peut également utiliser un "slice" (ou *tranche* en français, hors programme mais parfois très utile)

```
k = 3
k_plus_proches_voisins = distances_voisins_triee[0:k]  # récupère les k premiers éléments de distances_voisins_triee
k_plus_proches_voisins
```

On trouve, ce que l'on peut facilement vérifier sur le graphique, que :

- les trois points les plus proches de MM sont les points $D(2,3)$D(2,3), $B(2,2)$B(2,2) et $A(1,1)$A(1,1) ;
- la classe majoritaire parmi ces voisins est `'Bleu'` (2 contre 1) donc on peut prédire que le point cible MM est de classe "Bleue".

BILAN : écriture d'une fonction

Question 5 : Réunissez ces trois parties pour créer une fonction `kppv(donnees, cible, k)` qui prend en paramètres la table de données `donnees`, la donnée `cible` et le nombre `k` de voisins choisis.

Un test est proposée ci-dessous pour vérifier votre fonction.

```
# à vous de jouer !


# TEST
donnees = [A, B, C, D, E, F, G, H, I]
cible = {'nom': 'M', 'abs': 1, 'ord': 3}
k = 3
assert kppv(donnees, cible, k) == [{'nom': 'D', 'abs': 2, 'ord': 3, 'classe': 'Rouge', 'distance': 1.0},
                                   {'nom': 'B', 'abs': 2, 'ord': 2, 'classe': 'Bleu', 'distance': 1.4142135623730951},
                                   {'nom': 'A', 'abs': 1, 'ord': 1, 'classe': 'Bleu', 'distance': 2.0}]
```

Question 6 : Utilisez la fonction `kppv` pour vérifier la réponse à la question 2 de l'exercice 1.

```
# à compléter (q°2 : trois plus proches voisins de N)
```

Question 7 : Utilisez la fonction `kppv` pour vérifier la réponse à la question 3 de l'exercice 1.

```
# à compléter (q°3 : valeur de k pour que les k plus proches voisins de N soient majoritairement bleus)
```

Avec les distances de Manhattan et de Tchebychev

Question 8 : Adaptez la fonction `kppv(donnees, cible, k)` pour créer deux fonctions `kppv_man(donnees, cible, k)` et `kppv_tch(donnees, cible, k)` qui renvoie la liste des kk plus proches voisins avec les distances de Manhattan et de Tchebychev respectivement.

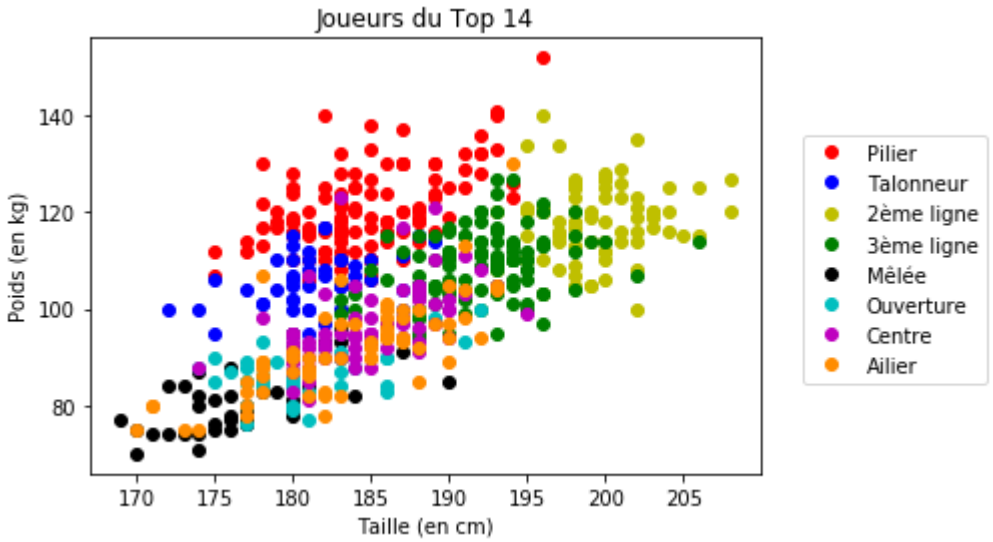
Question 9 : Trouve-t-on les mêmes kk plus proches voisins du point $N(4,4)$N(4,4) avec ces deux distances ?

Réponse :

Exercice 4 : TOP 14

Présentation

Vous allez travailler avec le fichier `top14.csv` qui contient des données sur tous les joueurs du top 14 de rugby. On peut représenter (une partie de) ces données de la façon suivante.



L'objectif de l'exercice est de prédire le poste le plus adéquat pour un nouveau joueur en fonction de son poids et de sa taille.

Représentation des données

On peut mémoriser les données du fichier `top14.csv` dans une table `joueurs` où chaque joueur est représenté par un dictionnaire.

```
import csv
fichier = open('top14.csv', 'r', encoding = 'UTF-8')
t = csv.DictReader(fichier, delimiter=',')
joueurs = [dict(ligne) for ligne in t] # création et construction du tableau par compréhension
fichier.close()
```

Voici le contenu de la table.

```
joueurs
```

Question 1 : Quelles sont les données fournies dans cette table ?

Réponse :

Question 2 : Quelle instruction permet d'accéder au poste du premier joueur de la table ?

```
# à compléter
```

Question 3 : Quelle instruction permet d'accéder au poids du deuxième joueur de la table ?

```
# à compléter
```

Question 4 : Quelle instruction permet d'accéder à la taille du troisième joueur de la table ?

```
# à compléter
```

On peut facilement représenter les différents joueurs dans un repère en fonction de leur taille (en abscisse) et leurs poids (en ordonnée) grâce à la bibliothèque `matplotlib`.

```
import matplotlib.pyplot as plt

taille = [int(j['Taille']) for j in joueurs] # construction des abscisses avec la taille
poids = [int(j['Poids']) for j in joueurs] # construction du tableau des valeurs d'attaque
plt.plot(taille, poids, 'ro') # construction du graphique
plt.title("Joueurs du Top 14") # légende
plt.xlabel('Taille (en cm)')
plt.ylabel('Poids (en kg)')
plt.show()  # affichage du graphique
plt.close()  # pour que les constructions ultérieures se fassent dans un nouveau graphique
```

On peut maintenant colorer ces points selon les postes des joueurs (Pilier, Talonneur, 2ème ligne, 3ème ligne, Mêlée, Ouverture, Centre, Ailier).

```
taille_piliers = [int(j['Taille']) for j in joueurs if j['Poste'] == 'Pilier']
poids_piliers = [int(j['Poids']) for j in joueurs if j['Poste'] == 'Pilier']

taille_talonneurs = [int(j['Taille']) for j in joueurs if j['Poste'] == 'Talonneur']
poids_talonneurs = [int(j['Poids']) for j in joueurs if j['Poste'] == 'Talonneur']

taille_2lignes = [int(j['Taille']) for j in joueurs if j['Poste'] == '2ème ligne']
poids_2lignes = [int(j['Poids']) for j in joueurs if j['Poste'] == '2ème ligne']

taille_3lignes = [int(j['Taille']) for j in joueurs if j['Poste'] == '3ème ligne']
poids_3lignes = [int(j['Poids']) for j in joueurs if j['Poste'] == '3ème ligne']
```

```
taille_melee = [int(j['Taille']) for j in joueurs if j['Poste'] == 'Mêlée']
poids_melee = [int(j['Poids']) for j in joueurs if j['Poste'] == 'Mêlée']

taille_ouvertures = [int(j['Taille']) for j in joueurs if j['Poste'] == 'Ouverture']
poids_ouvertures = [int(j['Poids']) for j in joueurs if j['Poste'] == 'Ouverture']

taille_centres = [int(j['Taille']) for j in joueurs if j['Poste'] == 'Centre']
poids_centres = [int(j['Poids']) for j in joueurs if j['Poste'] == 'Centre']

taille_ailiers = [int(j['Taille']) for j in joueurs if j['Poste'] == 'Ailier']
poids_ailiers = [int(j['Poids']) for j in joueurs if j['Poste'] == 'Ailier']

plt.plot(taille_piliers, poids_piliers, 'ro') # construction des piliers
plt.plot(taille_talonneurs, poids_talonneurs, 'bo') # construction des talonneurs
plt.plot(taille_2lignes, poids_2lignes, 'yo') # construction des 2èmes lignes
plt.plot(taille_3lignes, poids_3lignes, 'go') # construction des 3èmes lignes
plt.plot(taille_melee, poids_melee, 'ko') # construction des demis de mêlée
plt.plot(taille_ouvertures, poids_ouvertures, 'co') # construction des demis d'ouverture
plt.plot(taille_centres, poids_centres, 'mo') # construction des centres
plt.plot(taille_ailiers, poids_ailiers, color='darkorange', linestyle='none', marker = 'o') # construction des ailiers

plt.title("Joueurs du Top 14")
plt.xlabel('Taille (en cm)')
plt.ylabel('Poids (en kg)')
plt.legend(['Pilier', 'Talonneur', '2ème ligne', '3ème ligne', 'Mêlée', 'Ouverture', 'Centre', 'Ailier'], loc='upper center', bbox_t
o_anchor=(1.2, 0.8))
plt.show()
plt.close()
```

Prédire un poste

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 kk plus proches voisins pour lui indiquer quel poste est le plus proche de ses caractéristiques physiques.

Question 5 : Quelle est le dictionnaire `cible` de notre donnée cible ?

```
# à compléter
```

Question 6 : Adaptez la fonction `distance_euclidienne(d1, d2)` afin qu'elle renvoie la distance entre deux dictionnaires `d1` et `d2` par rapport à leurs clés `Taille` et `Poids`.

```
def distance_euclidienne(d1, d2):
    # à compléter
```

Question 7 : Utilisez la fonction `kppv` de l'exercice précédent pour répondre au problème. Essayez pour différentes valeurs de kk et pensez à vérifier la cohérence sur le graphique. *Vous devez un poste majoritaire qui est 'Talonneur' pour les premières valeurs de* kk.