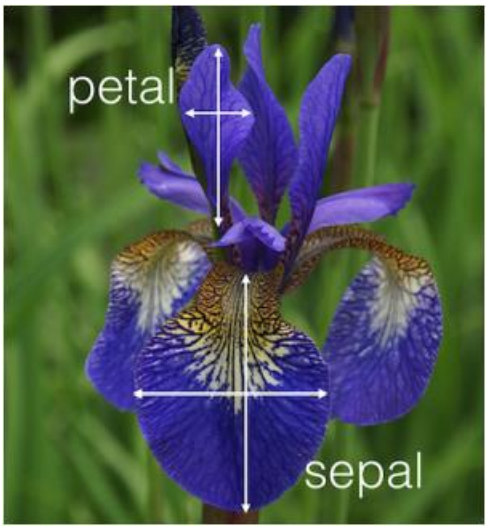
Algorithme des K plus proches voisins

I. Préparation de la base de données (Dataset)

1. Présentation de la base :

Pour mettre en œuvre cet algorithme, nous allons utiliser une base de données très connue sur les fleurs d'Iris. Cette base de données contient 150 mesures différentes des caractéristiques de la fleur et les classe en trois variétés :

Chaque mesure contient 4 caractéristiques ($features$) et est annotée par une étiquette ($label/target$).



Caractéristiques				Etiquette
Longueur sépale	Largeur sépale	Longueur pétale	Largeur pétale	Variété
5.1	3.5	1.4	0.2	0 (Setosa)
4.9	3.0	1.4	2.0	0 (Setosa)
7.0	3.2	4.7	1.4	1 (Versicolor)
6.4	3.2	4.5	1.5	1 (Versicolor)
6.3	3.3	6.0	2.5	2 (Virginica)
5.8	2.7	5.1	1.9	2 (Virginica)
7.1	3.0	5.9	2.1	2 (Virginica)

Nota :

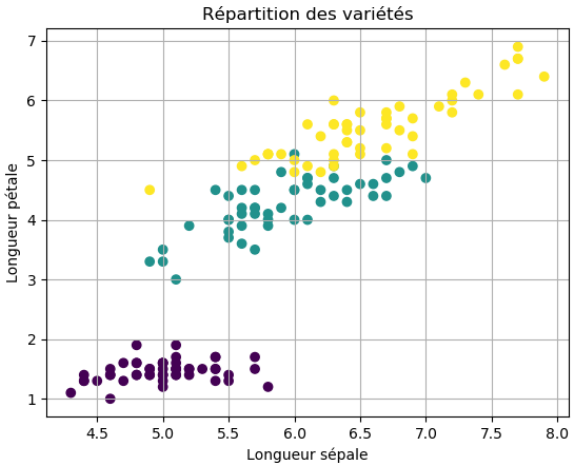
- Les dimensions sont données en cm.
- Les variétés Setosa, Versicolor, et Virginica sont respectivement étiquetée 0, 1, et 2.

2. Préparation du travail sous python.

- Importer les bibliothèques utiles pour ce TD :
  import numpy as np
  import matplotlib.pyplot as plt
- Copier le fichier $iris.txt$ dans le répertoire où vous souhaitez enregistrer votre code python.
- Ouvrir ce fichier à l'aide de la commande $loadtxt$ proposée par $numpy$.
  iris=np.loadtxt('iris.txt' , delimiter=',' , skiprows=1)

3. affichage de la base de données

Il est possible de représenter assez simplement la répartition des variétés en fonction de la longueur du sépale et de la longueur du pétale.



- On commence par définir les grandeurs que l'on souhaite relier :
  X1=iris[:,0] # Colonne "Longueur sépale"
  X2=iris[:,2] # Colonne "Longueur pétale"
  V=iris[:,4] # Colonne "Variété"

- Ensuite l'astuce consiste à associer une couleur à chaque variété :
  plt.scatter(X1,X2, c=V)

Quelques remarques :

- Ce graphique est intéressant car on peut voir en un coup d'œil comment sont réparties les trois classes (variétés).
- Une des classes est clairement éloignée des autres, il sera donc facile de prédire si une nouvelle donnée appartient à cette classe ou non. Par contre une mesure telle que :
  $$Longueur\ sépale \approx 6{,}2 \text{ et } Longueur\ pétale \approx 4{,}8$$
  serait difficile à classer avec la seule connaissance de ces deux caractéristiques.
- Cependant, il ne faut pas perdre de vue que cette représentation est partielle. On a en réalité 4 caractéristiques disponibles. Il est fort possible que la connaissance des deux autres caractéristiques permette de trancher.

4. Rangement aléatoire de la base de données :

Si on observe le tableau de données iris (par exemple en tapant *print(iris)* dans la console), on peut remarquer que les fleurs sont triées par variété (dernier élément de chaque ligne). Avant d'utiliser ces données pour faire des prédictions, il convient de les répartir aléatoirement.

On va donc avoir besoin d'utiliser le module $random$ de $numpy$.
Nos échantillons de données vont être choisis aléatoirement, mais une fois choisis, on souhaite les conserver en l'état pour les phases de test. En particulier, on ne veut pas qu'ils soient modifiés à chaque exécution du programme. La méthode $seed$ permet « d'initialiser » le module $random$ et donc de générer un aléatoire reproductible.

- Insérer la ligne de code suivante :
  np.random.seed(3)

5. Scission de la base de données.

Q1.a Ecrire une fonction $scission\ (T)$ qui prend en argument un tableau de données T et qui renvoie deux tableaux :

- $Train\_Set$: Ce premier tableau contiendra 80% des données prises aléatoirement dans T et servira à « entraîner » la machine.
- $Test\_Set.$: Ce second tableau contiendra les 20% de données restantes et servira ultérieurement à vérifier la fiabilité des prédictions proposées par la machine.

Nota :

En premier lieu dans cette fonction, il pourra être judicieux d'utiliser la méthode $random \cdot shuffle()$de $numpy$[1], permettant de changer aléatoirement la position des lignes d'un tableau (ou d'une liste). Attention le tableau initial est alors modifié de manière irréversible.

Q1b. Appliquer ensuite la fonction $scission$ à la base de données **iris** afin de définir deux tableaux $Iris\_Train\_Set$ et $Iris\_Test\_Set$ dont on se servira dans la suite.

II. Calcul des distances

Q2. Ecrire une fonction $distance(M,N)$ qui prend en argument les coordonnées (généralisées à n dimensions) de deux points M et N et qui renvoie la distance euclidienne entre ces deux points.

En général le nombre de classes est très petit par rapport au nombre d'éléments de notre échantillon de données. La colonne associée aux étiquettes (variétés dans notre exemple) contient donc un très grand nombre de doublons. Il peut être utile d'isoler seulement les classes, expurgées des doublons.

III. Détermination de l'occurrence dominante dans une liste

Q3. Ecrire une fonction $classes(T)$ qui prend en argument un tableau T (ou une liste) et qui renvoie la liste des éléments distincts (classes) contenus dans T.

Dans le cas de notre exemple, la commande $classes(iris[:,4])$ doit renvoyer [0,1,2] (mais pas forcément dans cet ordre).

Q4. Ecrire une fonction $dominant(L)$ qui prend en argument une liste L, et qui renvoie l'élément dont l'occurrence est la plus grande dans cette liste.

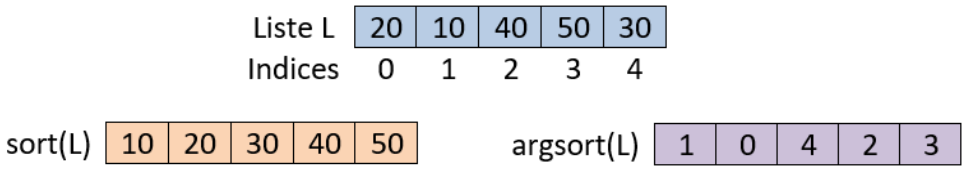
Vous pourrez utiliser la méthode $count$ qui permet de compter le nombre d'occurrence d'un élément e dans une liste L, méthode dont la syntaxe est la suivante : $L.count(e)$

IV. Mise en œuvre de l'algorithme KNN

La méthode $sort$ de $numpy$ permet de trier des listes ou des tableaux (type array).

La méthode $argsort$ de $numpy$ renvoie l'ordre dans lequel devraient se trouver les indices pour que la liste (ou le tableau) soit triée.

Dans l'exemple ci-dessous, pour que la liste soit triée, il faudrait placer en premier l'élément d'indice 1 (c'est-à-dire le nombre 10), puis l'élément d'indice 0 (le nombre 20), puis l'élément d'indice 4 (le nombre 30), puis l'élément d'indice 2 (le nombre 40) puis enfin, l'élément d'indice 3 (le nombre 50).



[1] https://www.cours-gratuit.com/tutoriel-python/tutoriel-python-list-comment-mlanger-alatoirement-une-liste-dlments

Q5. Ecrire une fonction $KNN(K, Train\_Set, New\_Data)$ qui prend en arguments un entier positif K, une base de données $Train\_Set$, une nouvelle observation New_Data et qui renvoie l'étiquette dominante pour les K plus proches voisins de cette nouvelle observation.

Vous pourrez vous aider de l'algorithme présenté à la page précédente, et utiliser la méthode $argsort$ de $numpy$.

Q6. Tester votre programme dans le cas particulier où :

- $K = 5$.
- $Train\_Set$: L'échantillon de données destiné à entraîner la machine sera $Iris\_Train\_Set$,
- New_Data sera constituée des 4 caractéristiques (features) de la première ligne du tableau $Iris\_Test\_Set$.

Vérifier, en particulier, si la prédiction renvoyée par la machine correspond bien à l'étiquette associée à cette ligne.

V matrice de confusion :

Il est possible de juger l'efficacité de l'algorithme à l'aide d'un outil appelé **matrice de confusion**. Les lignes de cette matrice renseignent le nombre d'éléments de chaque classe réellement contenus dans l'échantillon de test, et les colonnes le nombre d'éléments prédits dans chaque classe.
Par exemple, dans la matrice ci-contre :

		Nombre d'éléments prédits dans chaque classe		
		0 Setosa	1 Versicolor	2 Virginica
Nombre d'éléments réellement contenus dans chaque classe	0 Setosa	10	1	0
	1 Versicolor	0	8	1
	2 Virginica	0	0	10

Sur 11 données du type 0 (Setosa), 10 ont bien été prédites comme telle, mais une a été a été classée à tort dans la catégorie 1 (Versicolor).

Par contre toutes les prédictions sur les données du type 2 (Virginica) s'avèrent exactes.

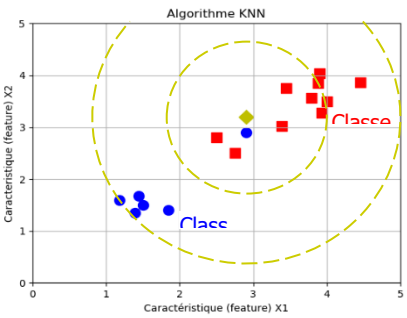
Q7. A l'aide d'une boucle for, et pour $K = 5$, réaliser une prédiction pour chacune des données du tableau $Iris\_Test\_Set$. Vous enregistrerez ces prédictions dans une liste Lp.

Q8. A partir des étiquettes (label) du tableau $Iris\_Test\_Set$ et de la liste des prédictions Lp, afficher la matrice de confusion M_c obtenue dans le cas où $K = 5$.

Le principe est le suivant : Si on a identité entre la prédiction P et la valeur test V, on incrémente le terme de la matrice $M_c[V, V]$, sinon on incrémente le terme $M_c[V, P]$.

VI. Influence de l'hyper paramètre K

Afin de trouver la valeur optimale de K, on va tester chaque ligne de notre tableau $Iris\_Test\_Set$ pour différentes valeurs de K, et comptabiliser le nombre de réussite dans nos prédictions. La valeur de K optimale sera celle pour laquelle le nombre de réussites est maximal.



Q9. Tracer une courbe montrant l'évolution du taux de réussites dans nos prédictions pour $K \in [1,2,\ldots,100]$. Expliquer l'allure de la courbe.

Pour rappel, sur les 150 mesures de la base de données **Iris**, 80% d'entre elles (soit 120) sont utilisées pour entraîner l'algorithme et les 20% restantes (soit 30) sont utilisées pour tester l'algorithme.

VI Pour aller plus loin : Scikit-Learn :

$Scikit-Learn$ est un outil professionnel dans lequel les modèles d'apprentissage automatique (machine learning) ont été implémentés. Ces modèles peuvent être de nature différente (régression linéaire, régression polynomiale, arbre de décision, KNN, réseau de neurones, etc.), mais leur utilisation est toujours la même :

1. On commence par sélectionner un modèle et préciser ses hyper-paramètres :

$$model = KNeighborsClassifier(n\_neighbors = 5)$$

2. Ensuite, on entraine le modèle sur des données.

Ici, les données sont divisées en deux tableaux $numpy$ $(X\_Train\_Set, y\_Train\_Set)$.

- X_Train_Set correspond aux caractéristiques de l'échantillon d'entraînement
- y_Train_Set aux étiquettes :

$$model.fit(X\_Train\_Set, y\_Train\_Set)$$

Attention ! X et y doivent avoir deux dimensions (au besoin, utiliser $reshape(n, 1)$ pour y)

3. Puis on évalue l'efficacité du modèle sur l'échantillon de test :

$$model.score(X\_Test\_Set, y\_Test\_Set)$$

4. Enfin on peut utiliser le modèle pour faire des prédictions :

$$model.predict(X\_New\_Data)$$

Copier puis tester les lignes de code suivantes :

```
"""Prédictions avec Scikit-Learn"""
from sklearn.neighbors import KNeighborsClassifier

# Selection du modèle
model=KNeighborsClassifier(n_neighbors=5)

# Entraînement du modèle
X_Train=Iris_Train_Set[:,:-1] # Caractéristiques de l'échantillon
d'entraînement
y_Train=Iris_Train_Set[:,-1] # Etiquettes de l'échantillon
d'entraînement

model.fit(X_Train,y_Train) #On entraine le modèle

# Evaluation du modèle
X_Test=Iris_Test_Set[:,:-1] # Caractéristiques de l'échantillon de
test
y_Test=Iris_Test_Set[:,-1] # Etiquettes de l'échantillon de test

a=model.score(X_Test,y_Test) # On calcule l'efficacité du modèle sur
l'échantillon de test
print('Le modèle est efficace à', a*100,'%')

# Prédictions
print(model.predict(X_Test))
```

Une fois que les prédictions sont effectuées, il est possible de les évaluer. Par exemple, pour afficher la matrice de confusion associée à nos données, il suffit d'écrire les lignes de code suivantes :

```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_Test,model.predict(X_Test)))
```

Q10. Afficher la matrice de confusion dans le cas $K = 5$. Comparer à celle obtenue en Q8.

Si on veut prédire la classe d'une nouvelle donnée non tabulée, il suffit d'entrer les lignes de code suivantes :

```
X_New_Data=np.array([6.4,3.2,4.5,1.5])
X_New_Data =X_New_Data.reshape(1,4) # Rappel : X doit avoir deux
dimensions
print(model.predict(X_New_Data))
```

On souhaite maintenant tracer l'évolution de l'efficacité du modèle KNN en fonction de l'hyper-paramètre K.

Q11. A l'aide d'une boucle for calculer l'efficacité du modèle KNN pour $K \in [1,2,\ldots,100]$, puis tracer la courbe représentative de cette évolution. Comparer à la courbe obtenue en Q9.