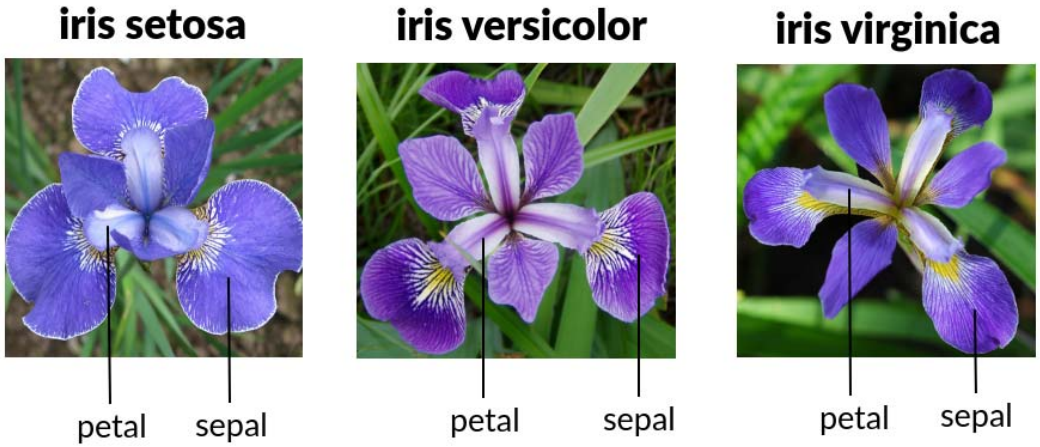
Exercice guidé sur la classification d'iris - Apprentissage automatique supervisé

Nous allons travailler sur le jeu de données des iris de Fischer, utilisé très couramment comme base de travail pour découvrir l'apprentissage automatique supervisé et non supervisé.

Pour en savoir plus sur ce jeu de données : https://fr.wikipedia.org/wiki/Iris_de_Fisher (https://fr.wikipedia.org/wiki/Iris_de_Fisher)

Dans cet exercice guidé, nous allons chercher à déterminer à quelle variété un iris donné appartient, en fonction de ses caractéristiques, en utilisant l'algorithme des k plus proches voisins.
Chaque iris est caractérisé par 4 données : les longueurs, et largeurs de la sépale et de la pétale.



Nous allons utiliser Scikit-learn qui est une bibliothèque libre Python destinée à l'apprentissage automatique et qui comportent, entre autres des jeux de données dont celui des iris de Fischer.

Le script suivant charge les données :

Entrée[2]:
```
from sklearn.datasets import load_iris
import numpy as np
iris = load_iris()
```

Description des données

`iris` obtenu dans le script précédent est un dictionnaire tel que :

- `iris['data']` est une matrice `numpy` dont chaque ligne contient les caractéristiques d'un iris : longueur, largeur de la sépale et de la pétale.

sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
5.1	3.5	1.4	0.2
4.9	3.0	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5.0	3.6	1.4	0.2

- `iris['target']` est un vecteur contenant les variétés d'iris : setosa (0), versicolor (1) ou virginica (2).

target
0
0
0
0
0

L'objectif est donc, à partir des caractéristiques d'un iris, de déterminer sa variété.

Remarque : pour accéder à une clé `k` dans un dictionnaire `d` , on peut écrire `d.k` plutôt que `d['k']` .
On utilisera donc `iris.data` et `iris.target` dans la suite.

Prise en main du jeu de données

Affichage des dimensions de la matrice qui contient les caractéristiques des iris du jeu de données iris.

Pour le nombre de lignes (qui correspond au nombre d'iris) : 150

Pour le nombre de colonnes (qui correspond au nombre de caractéristiques) :4

Pour la manipulation de tableaux numpy_.array, vous pouvez, entre autres, consulter le site:

https://courspython.com/tableaux-numpy.html (https://courspython.com/tableaux-numpy.html)

Entrée[3]:
```
np.shape(iris.data)
# il y a 150 iris, chacun ayant 4 caractéristiques
```

Sortie[3]: `(150, 4)`

Affichage des caractéristiques du 1er iris du jeu de données:

Entrée[4]:
```
iris.data[0]
# pour obtenir les caracteristiques du 1er iris
```

Sortie[4]:
```
array([5.1, 3.5, 1.4, 0.2])
```

- Affichage de l'étiquette (ou label ou classe) du 1er iris :

Entrée[5]:
```
iris.target[0]
# l'étiquette du 1er iris : 0 donc setosa
```

Sortie[5]:
```
0
```

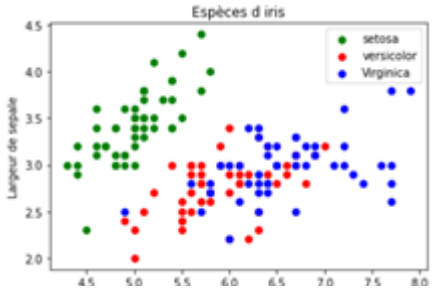
Le premier iris du jeu de données a donc :

- une longueur de sépale de 5.1 cm,
- une largeur de sépale de 3.5 cm,
- une longueur de pétale de 1.4 cm
- et une largeur de pétale de 0.2 cm.

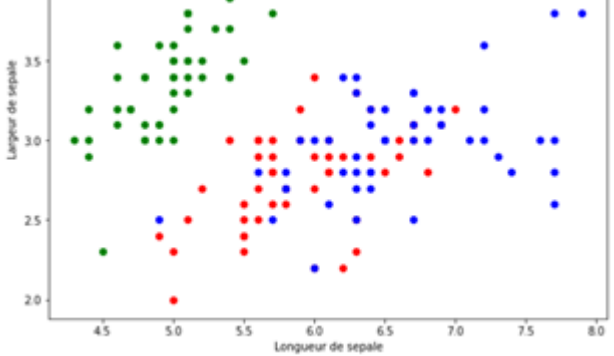
Il est de la variété setosa.

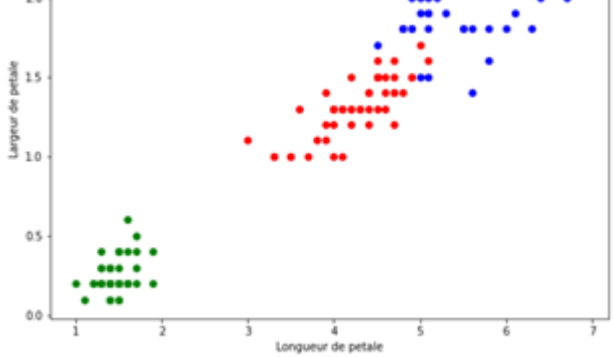
Tracé de graphiques

Les données sont donc des points de $\mathbb{R}^4$. Pour les visualiser, on peut regarder leurs projections en tracant d'abord les 2 premiers axes (longueur et largeur de la sépale),



et sur les deux autres axes (longueur et largeur de la pétale) :


Largeur de sepale
Longueur de sepale
2.0
2.5
3.0
3.5
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


Largeur de petale
Longueur de petale
0.0
0.5
1.0
1.5
1
2
3
4
5
6
7

Entrée[6]:

```
import matplotlib.pyplot as plt

X1_setosa =[]
X2_setosa = []
X1_versicolor= []
X2_versicolor= []
X1_virginica = []
X2_virginica = []

# Création des listes avec les 2 1eres caracteristiques de chaque classe (0,1,2):
for i in range( len(iris.target) ): # recupération de chaque iris
  liris = iris.data[i]
  classe = iris.target[i]

  if classe == 0 :
    X1_setosa.append(liris[0])
    X2_setosa.append(liris[1])
  elif classe ==1:
    X1_versicolor.append(liris[0])
    X2_versicolor.append(liris[1])
  else :
    X1_virginica.append(liris[0])
    X2_virginica.append(liris[1])

# Tracé du graphique:
plt.title('Espèces d iris')
plt.xlabel('Longueur de sepale')  # titre de l'axe des abscisses
plt.ylabel('Largeur de sepale')  # titre de l'axe des ordonnées

"""  #version avec plot
plt.plot(X1_setosa,X2_setosa, marker='o', color='g', markersize=3, linestyle='', label="Setosa")
plt.plot(X1_versicolor,X2_versicolor, marker='o', color='r', markersize=3, linestyle='', label="Versicolor")
plt.plot(X1_virginica,X2_virginica, marker='o', color='b', markersize=3, linestyle='', label="Virginica")
plt.legend()
plt.show()
"""
#version avec scatter
plt.scatter(X1_setosa,X2_setosa, color='g', label='setosa')
plt.scatter(X1_versicolor,X2_versicolor, color='r', label='versicolor')
plt.scatter(X1_virginica,X2_virginica, color='b', label='Virginica')
plt.legend()
plt.show()
```

Afficher les

Entrée[1]:

```
X_setosa = [ [],[],[],[] ]
X_versicolor= [ [],[],[],[] ]
X_virginica= [ [],[],[],[] ]

# Création des listes avec les 2 1eres caracteristiques de chaque classe (0,1,2):
for i in range( len(iris.target) ): # recupération de chaque iris
    liris = iris.data[i]
    classe = iris.target[i]

    if classe == 0 :
        for j in range (4) :
          X_setosa[j].append(liris[j])
      elif classe ==1:
        for j in range (4) :
          X_versicolor[j].append(liris[j])
      else :
        for j in range (4) :
          X_virginica[j].append(liris[j])

# Tracé des graphiques:
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,7))
fig.suptitle('Espèces d iris')

ax1.set_xlabel('Longueur de sepale')  # titre de l'axe des abscisses
ax1.set_ylabel('Largeur de sepale')  # titre de l'axe des ordonnées
"""
ax1.plot(X_setosa[0],X_setosa[1], marker='o', color='g', markersize=3, linestyle='', label="Setosa")
ax1.plot(X_versicolor[0],X_versicolor[1], marker='o', color='r', markersize=3, linestyle='', label="Versicolor")
ax1.plot(X_virginica[0],X_virginica[1], marker='o', color='b', markersize=3, linestyle='', label="Virginica")
"""
ax1.scatter(X_setosa[0],X_setosa[1], color='g', label='setosa')
ax1.scatter(X_versicolor[0],X_versicolor[1],  color='r', label='versicolor')
ax1.scatter(X_virginica[0],X_virginica[1], color='b', label='Virginica')

ax1.legend()

ax2.set_xlabel('Longueur de petale')  # titre de l'axe des abscisses
ax2.set_ylabel('Largeur de petale')  # titre de l'axe des ordonnées
ax2.scatter(X_setosa[2],X_setosa[3], color='g', label='setosa')
ax2.scatter(X_versicolor[2],X_versicolor[3],  color='r', label='versicolor')
ax2.scatter(X_virginica[2],X_virginica[3], color='b', label='Virginica')
"""
ax2.plot(X_setosa[2],X_setosa[3], marker='o', color='g', markersize=3, linestyle='', label="Setosa")
ax2.plot(X_versicolor[2],X_versicolor[3], marker='o', color='r', markersize=3, linestyle='', label="Versicolor")
ax2.plot(X_virginica[2],X_virginica[3], marker='o', color='b', markersize=3, linestyle='', label="Virginica")
"""
```

```
ax2.legend()
plt.plot()
```

```
Traceback (most recent call last):
  File "<input>", line 6, in <module>
NameError: name 'iris' is not defined
```

Classification

Un algorithme de classification demande de séparer les données que l'on souhaite en 2 ensembles :

- les **données d'entraînement** (`X_train` dans la suite) que l'on utilise pour améliorer le modèle
- les **données de test** (`X_test`) pour juger des performances du modèle.

Séparation données d'entraînement / test

Créer une fonction `separer(X, Y, p)` qui permet de séparer l'ensemble des iris du jeu de données en 2, suivant la proportion p. La fonction va générer et rerouner 4 tableaux

- `X_train` qui contient les caractéristiques de chaque iris des données d'entraînement.
- `Y_train` contient la classe de chacun des iris (0 = setosa, 1 = versicolor, 2 = virginica).
- De même pour `X_test` et `Y_test` .

Remarque : Si ce n'est pas le cas, pensez à convertir les données générées en `numpy_.array_` pour pouvoir les manipuler plus facilement notamment pour les tracés à venir.

Tester la fonction pour qu'elle répartisse le jeu de données suivant une répartition 80% pour les données d'entrainement et 20% pour les données de test.

Entrée[6]:

```
def separer(X, Y, p):
    Xtrain, Xtest, Ytrain, Ytest = [], [], [], []
    for i in range(len(X)):
        if i <= p * len(X) :
            Xtrain.append(X[i])
            Ytrain.append(Y[i])
        else:
            Xtest.append(X[i])
            Ytest.append(Y[i])
    return np.array(Xtrain), np.array(Xtest), np.array(Ytrain), np.array(Ytest)

Xtrain, Xtest, Ytrain, Ytest = separer(iris.data, iris.target, .8) # cette séparation n'est pas deale avec notre jeu de donnée

# 2e version + adaptée au jeu de données iris:
# on met p*10 (p correspondant au %) iris sur 10 dans Xtrain , Ytrain, et 1/5 dans Xtest, Ytest dans l'ordre ou ils sont stocke
def separer2(X, Y, p):
    nb = p*10
    cptr = 0
    Xtrain, Xtest, Ytrain, Ytest = [], [], [], []
    for i in range(len(X)):
      if cptr <nb  :
      #if i%nb !=0 :  #
            Xtrain.append(X[i])
            Ytrain.append(Y[i])
            cptr +=1
      else:
            Xtest.append(X[i])
            Ytest.append(Y[i])
            cptr +=1
            if cptr == 10 :
              cptr = 0

    return np.array(Xtrain), np.array(Xtest), np.array(Ytrain), np.array(Ytest)

Xtrain, Xtest, Ytrain, Ytest = separer(iris.data, iris.target, .8)
Xtrain, Xtest, Ytrain, Ytest = separer2(iris.data, iris.target, .8)
```

Afficher les caractéristiques (longueur, largeur de sépale et de pétale) de la première fleur des données d'entraînement, ainsi que sa variété :

Entrée[7]:
```
print( Xtrain[0] ) # caractéristiques
print( Ytrain[0] ) # variété
```

```
[5.1 3.5 1.4 0.2]
0
```

4. Mise en œuvre de l'algorithme des k plus proches voisins

Calcul de distances : distance euclidienne

Soient $x = (x_0, x_1, x_2, x_3)$ et $y = (y_0, y_1, y_2, y_3)$ deux caractéristiques de fleurs (longueur et largeur de la sépale et de la pétale). On définit la distance euclidienne d entre ces fleurs par :

$$d(x, y) = \sqrt{(x_0 - y_0)^2 + (x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2}$$

- Définir la fonction `d(x, y)` qui prend en paramètres 2 vecteurs(tableaux) x et y et retourne la valeur de la distance euclidienne entre x et y.

Entrée[14]:
```
def d(x, y):
    return ((x[0] - y[0])**2 + (x[1] - y[1])**2 + (x[2] - y[2])**2 + (x[3] - y[3])**2)**0.5

def d( x, y ) : # calcul de distances pour des tailles de vecteurs quelconques
  s=0
  for i in range (0, len(x)):
    s = s + (x[i] - y[i])**2
  return s**0.5
```

Remarque : on peut aussi utiliser le fait que la distance euclidienne entre deux vecteurs X et Y est $\sqrt{(X - Y)^\top (X - Y)}$, ce qui est plus rapide à calculer. De plus, on ne s'intéresse qu'aux plus proches voisins, et pas à leur distance exacte. On peut donc se passer du calcul de la racine carrée,

- Définir la fonction `d2(x, y )` qui calcule la distance euclidienne suivant cette formule.

Entrée[15]:
```
def d2(x, y):
    z = x - y
    return z.T.dot(z)
```

- Définir la fonction `voisins(x, X, k, dist)` qui renvoie la liste des indices des k plus proches voisins de x dans X en utilisant la fonction dist de calcul de distance:

Entrée[16]:

```
def voisins(x, X, k, dist):

    listeDistanceIndices=[] # liste de listes[distance, indice]
    for i in range ( len(X)) :
      distance = dist( x, X[i])
      listeDistanceIndices.append([distance, i])
    listeDistanceIndices.sort()
    lesVoisins = [] # liste qui ne contiendra que les indices
    for i  in range( k ):
      lesVoisins.append( listeDistanceIndices[i][1] )
    return lesVoisins

k=9
x = Xtest[0]
V = voisins(x, Xtrain, k, d)


# variante en utilisant la fonction sorted
def voisins(x, X, k, dist): # renvoie les k plus proches voisins de x dans X
    indices = sorted(range(len(X)), key=lambda i: dist(x, X[i]))
    return indices[:k]

x = Xtest[0]
V = voisins(x, Xtrain, k, d)
```

- Fonction `majoritaire(L)` qui détermine et renvoie la classe majoritaire de la liste L, passée en paramètre.

Entrée[17]:

```
def majoritaire(L):
  compte = {} # compte[e] = nombre d'occurrences de e dans L
  for e in L:
        if e in compte:
            compte[e] += 1
        else:
            compte[e] = 1
  kmax = L[0]
  for k in compte:
        if compte[k] > compte[kmax]:
            kmax = k
  return kmax
# test de la fonction en affichant la classe majoritaire du jeu de données
print( majoritaire( iris.target) )

# version 2 en utilisant la fonction get des dictionnaires
def majoritaire(L): # renvoie la classe qui apparaît le plus souvent dans L
    compte = {}
    for e in L:
        compte[e] = compte.get(e, 0) + 1
    return max(compte, key=compte.get)
```

```
0
```

- A l'aide des fonctions précédentes, écrire la fonction `knn(x, X, Y, k, dist)` qui retourne la classe prédite pour x par l'algorithme des k plus proches voisins.

Entrée[18]:

```
def knn(x, X, Y, k, dist): # renvoie la classe prédite pour x par l'algorithme des k plus proches voisins
    # détermination des indices des k plus proches voisins
    lesVoisins = voisins(x, X, k, dist)

    # détermination des classes des k plus proches voisins
    lesClasses = [ Y[i] for i in lesVoisins ]

    # determination de la classe majoritaire des k voisins obtenues qui sera la classe de x
    laClasseMajoritaire = majoritaire( lesClasses )
    return laClasseMajoritaire
```

Ecrire la fonction `affichePrediction( x, classe_x , Xtrain , Ytrain)` qui affiche `x` et les `k` plus proches voisins de `x` ainsi que la classe prédite

Entrée[21]:

```
def affichePrediction( x, k , Xtrain , Ytrain ):

  X_setosa = [ [],[],[],[] ]
  X_versicolor= [ [],[],[],[] ]
  X_virginica= [ [],[],[],[] ]

  classe_x = knn(x, Xtrain, Ytrain , k, d)

  coul = ['g','r','b']
  neighbors_i = voisins(x, Xtrain, 7, d2)
  neighbors = Xtrain[neighbors_i]

  # Création des listes avec les 2 1eres caracteristiques de chaque classe (0,1,2):
  for i in range( len(Ytrain) ): # recupération de chaque iris
    liris  = Xtrain[i]
    classe = Ytrain[i]

    if classe == 0 :
      for j in range (4) :
        X_setosa[j].append(liris[j])
    elif classe ==1:
      for j in range (4) :
        X_versicolor[j].append(liris[j])
    else :
      for j in range (4) :
        X_virginica[j].append(liris[j])

  # Tracé des graphiques:
  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,7))
  fig.suptitle('Espèces d iris')

  ax1.set_xlabel('Longueur de sepale')  # titre de l'axe des abscisses
  ax1.set_ylabel('Largeur de sepale')  # titre de l'axe des ordonnées
  ax1.scatter(X_setosa[0],X_setosa[1], color='g', label='setosa')
  ax1.scatter(X_versicolor[0],X_versicolor[1],  color='r', label='versicolor')
  ax1.scatter(X_virginica[0],X_virginica[1], color='b', label='Virginica')

  for i in range(len( neighbors )):
    couleur = coul[ Ytrain[neighbors_i[i]] ]
    ax1.scatter(neighbors[i, 0], neighbors[i, 1], s=120,  color=couleur,  marker='X')

  ax1.scatter(x[0],x[1], s=200,color=coul[classe_x] )
  ax1.legend()

  ax2.set_xlabel('Longueur de petale')  # titre de l'axe des abscisses
  ax2.set_ylabel('Largeur de petale')  # titre de l'axe des ordonnées
```

```
  ax2.scatter(X_setosa[2],X_setosa[3], color='g', label='setosa')
  ax2.scatter(X_versicolor[2],X_versicolor[3],  color='r', label='versicolor')
  ax2.scatter(X_virginica[2],X_virginica[3], color='b', label='Virginica')

  for i in range(len( neighbors )):
    couleur = coul[ Ytrain[neighbors_i[i]] ]
    ax2.scatter(neighbors[i, 2], neighbors[i, 3],s=120,  color=couleur,  marker='X')

  ax2.scatter(x[2],x[3], s=200,color=coul[classe_x] )
  ax2.legend()

  ax2.legend()
  plt.show()

affichePrediction( x, k, Xtrain , Ytrain )
```

Fonction `predict(i, k, d)` qui renvoie la classe prédite pour le ième iris du jeu de donnée X_test

Entrée[ ]:
```
def predict(Xtest, Xtrain, i, k, d): # renvoie la classe prédite pour Xtest[i]
    return knn( Xtest[i], Xtrain, Ytrain, k, d )
```

Entrée[ ]:
```
print("Classe prédite pour X_test[0] :", predict(0, 3, d2))
print("Classe réelle pour X_test[0] :", Ytest[0])
```

```
Classe prédite pour X_test[0] : 0
Classe réelle pour X_test[0] : 0
```

Ecrire la fonction `affichePrediction( x, classe_x , Xtrain , Ytrain)` qui affiche `x` et les `k` plus proches voisins de `x` ainsi que la classe prédite :

Entrée[ ]:

```
def affichePrediction( x, k , Xtrain , Ytrain ):

  X_setosa = [ [],[],[],[] ]
  X_versicolor= [ [],[],[],[] ]
  X_virginica= [ [],[],[],[] ]

  classe_x = knn(x, Xtrain, Ytrain , k, d)

  coul = ['g','r','b']
  neighbors_i = voisins(x, Xtrain, 7, d2)
  neighbors = Xtrain[neighbors_i]

  # Création des listes avec les 2 1eres caracteristiques de chaque classe (0,1,2):
  for i in range( len(Ytrain) ): # recupération de chaque iris
    liris  = Xtrain[i]
    classe = Ytrain[i]

    if classe == 0 :
      for j in range (4) :
        X_setosa[j].append(liris[j])
    elif classe ==1:
      for j in range (4) :
        X_versicolor[j].append(liris[j])
    else :
      for j in range (4) :
        X_virginica[j].append(liris[j])

  # Tracé des graphiques:
  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,7))
  fig.suptitle('Espèces d iris')

  ax1.set_xlabel('Longueur de sepale')  # titre de l'axe des abscisses
  ax1.set_ylabel('Largeur de sepale')  # titre de l'axe des ordonnées
  ax1.scatter(X_setosa[0],X_setosa[1], color='g', label='setosa')
  ax1.scatter(X_versicolor[0],X_versicolor[1],  color='r', label='versicolor')
  ax1.scatter(X_virginica[0],X_virginica[1], color='b', label='Virginica')

  for i in range(len( neighbors )):
    couleur = coul[ Ytrain[neighbors_i[i]] ]
    ax1.scatter(neighbors[i, 0], neighbors[i, 1], s=120,  color=couleur,  marker='X')

  ax1.scatter(x[0],x[1], s=200,color=coul[classe_x] )
  ax1.legend()

  ax2.set_xlabel('Longueur de petale')  # titre de l'axe des abscisses
  ax2.set_ylabel('Largeur de petale')  # titre de l'axe des ordonnées
```

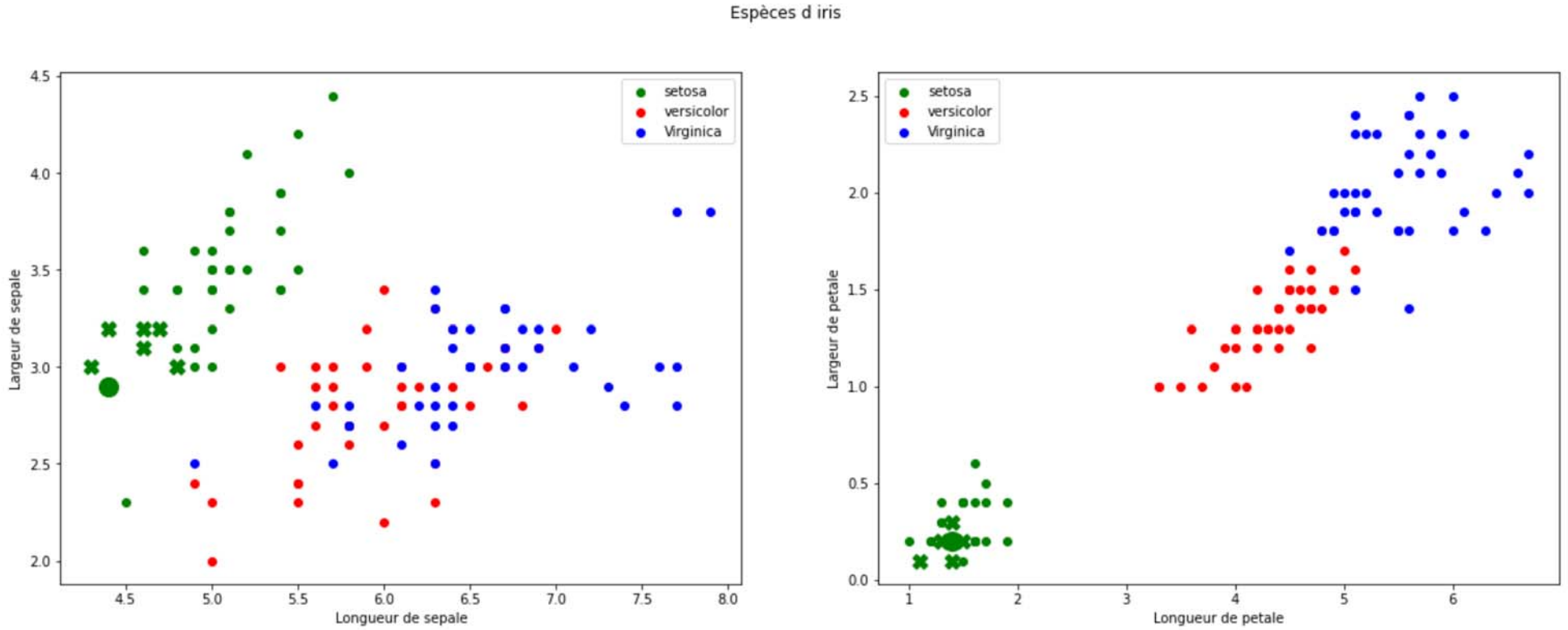
```
  ax2.scatter(X_setosa[2],X_setosa[3], color='g', label='setosa')
  ax2.scatter(X_versicolor[2],X_versicolor[3],  color='r', label='versicolor')
  ax2.scatter(X_virginica[2],X_virginica[3], color='b', label='Virginica')

  for i in range(len( neighbors )):
    couleur = coul[ Ytrain[neighbors_i[i]] ]
    ax2.scatter(neighbors[i, 2], neighbors[i, 3],s=120,  color=couleur,  marker='X')

  ax2.scatter(x[2],x[3], s=200,color=coul[classe_x] )
  ax2.legend()

  ax2.legend()
  #plt.show()

affichePrediction( x, k, Xtrain , Ytrain )
```



Évaluation du modèle

Ecrire la fonction `precision(Xtest, Ytest, Xtrain, Ytrain, k, dist)` qui calcule la précision càd le nombre de prédictions correctes sur le nombre de prédictions totales

Entrée[22]:
```
def precision(Xtest, Ytest, Xtrain, Ytrain, k, dist):

   # print (Xtest, Ytest)
    n = len(Xtest)
    p = 0
    for i in range(n):
        x = Xtest[i]
        classe_avec_knn= knn(x, Xtrain, Ytrain, k, dist)
        # print(classe)
        if classe_avec_knn == Ytest[i]: # la classe obtenue correspond-elle a la classe predite?
            p += 1
        # else:
        #   print(f"Erreur pour X_test[{i}] : {iris.target_names[predict(i, k)]} au lieu de {iris.target_names[Y_test[i]]}")
    return p/n
```

Entrée[23]:
```

```

Sortie[23]: 0.9666666666666667

Pour savoir quelle est la meilleure valeur de k, afficher la précision en fonction de k :

Entrée[24]:
```
def plot_precision(kmax, Xtest, Ytest, Xtrain, Ytrain, d):
    plt.figure()
    plt.plot(range(1, kmax), [precision(Xtest, Ytest, Xtrain, Ytrain,k, d) for k in range(1, kmax)])
    plt.xlabel("k")
    plt.ylabel("Précision")
    plt.show()
plot_precision(50, Xtest, Ytest, Xtrain, Ytrain, d2)
```

Zoomons sur $k \in [1, 15]$:

Entrée[ ]:
```
plot_precision(15, Xtest, Ytest, Xtrain, Ytrain, d2)
```

Graphiquement, la précision est maximale pour $k = 9$:

Entrée[25]:
```
precision(Xtest, Ytest, Xtrain, Ytrain, k, d2)
```

Sortie[25]: 0.9666666666666667

Creer et afficher la matrice de confusion pour $k = 9$:

Entrée[26]:
```
k=9
M = np.zeros((3, 3), dtype=int)
for i in range(len(Xtest)):
  x = Xtest[i]
  cl = knn(x, Xtrain, Ytrain, k, d)
  M[cl][Ytest[i]] += 1
print(M)
```

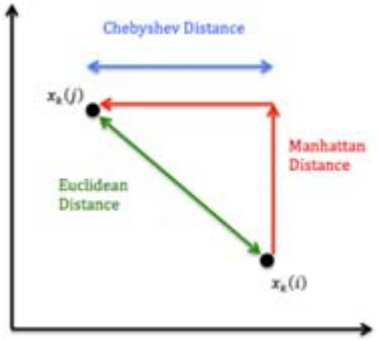
```
[[10  0  0]
 [ 0 10  1]
 [ 0  0  9]]
```

Entrée[ ]:
```

```

Autres distances

Essais d'autres distances pour comparer la précision :



Distance de Manhattan

Créer la fonction `distManhattan` qui calcule la distance de Manhattan pour les vecteurs x et y.

Entrée[27]:
```
def distManhattan(x, y):
    return sum(abs(x[i] - y[i]) for i in range(len(x)))
```

Calculer la précision en utilisant `distManhattan` .

Entrée[28]:
```
precision(Xtest, Ytest, Xtrain, Ytrain, k, distManhattan )# on obtient la même précision qu'avec la distance euclidienne
```

Sortie[28]: `0.9666666666666667`

Distance de Thebychev

Créer la fonction `distTchebychev` qui calcule la distance de Tchebytchev pour les vecteurs x et y.
Calculer la précision en utilisant `distTchebychev` .

Entrée[1]:
```
def distThebychev(x, y):
    return max(abs(x[i] -y[i]) for i in range(len(x)))
```

Entrée[31]:
```
precision(Xtest, Ytest, Xtrain, Ytrain, k, distThebychev)
```

Sortie[31]: `1.0`

Conclusion : l'utilisation d'autres distances n'a pas permis d'améliorer la précision.