kppv Revision Iris

In [2]:
```
# Tous les imports du notebook, cellule à exécuter dès le début
from math import sqrt
from random import shuffle
import matplotlib.pyplot as plt
import csv
```

Thème 5 : Algorithmique

16

Algorihtme des k-plus proches voisins : Application : classification des Iris.

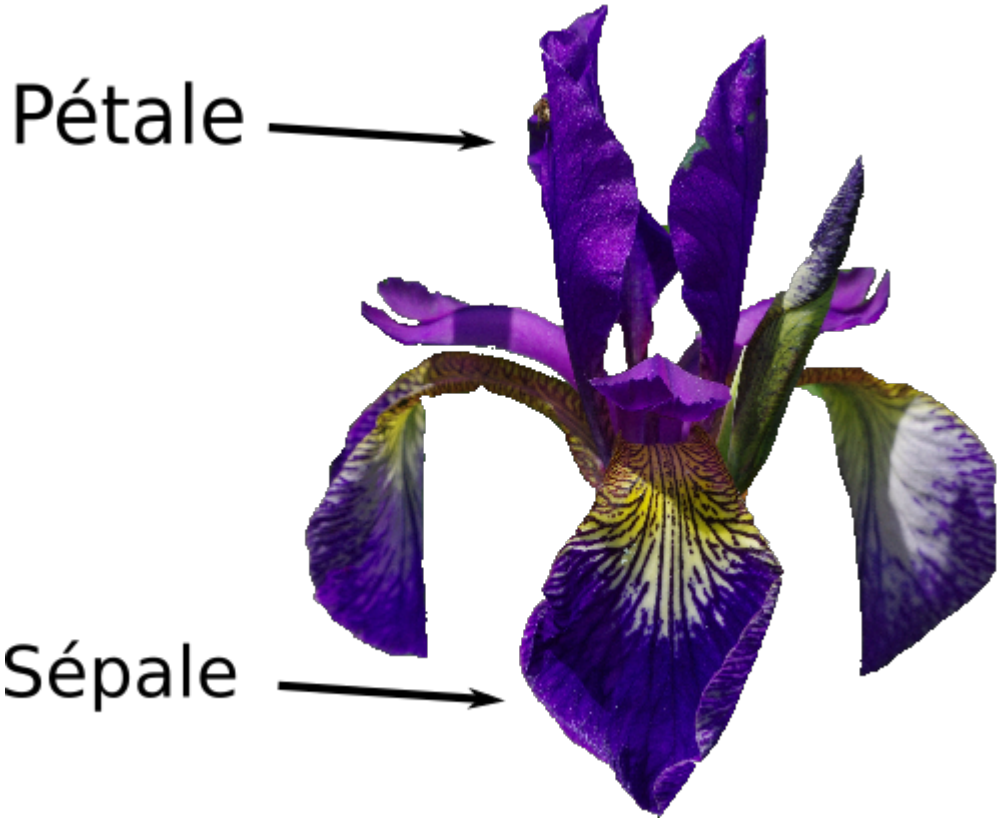
En 1936, Edgar Anderson a collecté des données sur 3 espèces d'iris : "iris setosa", "iris virginica" et "iris versicolor" our chaque iris étudié, Anderson a mesuré (en cm) :

- la largeur des sépales
- la longueur des sépales
- la largeur des pétales
- la longueur des pétales

Iris setosa

Iris virginica

Iris versicolor



Diverses mesures

Les données sont ici stockées dans un fichier au format `.csv` , dont voici les deux premières lignes :

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
```

Chaque ligne de données est composée des descripteurs et d'une étiquette (l'espèce de l'iris) , séparés par une virgule.

Pour chacun de ces individus on dispose des champs :

```
- longueur_sepale : longueur des sépales

- largeur_sepale  : largeur des sépales

- longueur_petale : longueur des pétales

- largeur_petale  : largeur des pétales

- espece          : 'Iris-setosa', 'Irid_virginica' ou 'Iris-versicolor'
```

0.1. Durant ce TP, vous allez chercher à déterminer les espèce d'iris à partir des mesures des pétales et sépales.

On va donc chercher, à partir des quatre descripteurs longueur_sepale, largeur_sepale, longueur_petale et largeur_petale, à effectuer une prédiction de l'étiquette espece.

1. Chargement des données

In [10]:
```
def lirefichier(nomfichier):
    fichierCSV = open(nomfichier,"r")
    lignes = csv.reader(fichierCSV)
    next(lignes)
    data = list(lignes)

    for i in range(len(data)):
        for j in range(len(data[i])-1):
            data[i][j] = float(data[i][j])

    return data

A=(2.5,0.75)

dataset=lirefichier('iriscomplet.csv')
print(dataset[0])
```

```
[5.1, 3.5, 1.4, 0.2, 'Iris-setosa']
```

1.2. Exercice 1

1. Quel est le type de données de la variable `dataset`
2. De quels types sont les 5 éléments stockés dans `dataset[0]` ?

1.2.1. Réponses :

1.
2.

2. Affichage des données

Dans notre exemple, nous avons 4 variables descriptives.
Or il n'est pas possible de tracer un graphique avec 4 axes.
Une solution est de représenter les données en choisissant deux axes parmi les variables descritives, cela donne :

	Abscisse	Ordonnée
Graphique 1	largeur des sépales	longueur des sépales
Graphique 2	largeur des sépales	largeur des pétales
Graphique 3	largeur des sépales	longueur des pétales
Graphique 4	longueur des sépales	largeur des pétales
Graphique 5	longueur des sépales	longueur des pétales
Graphique 6	largeur des pétales	longueur des pétales

Voici, ci-dessous, le résultat obtenu.

Notes :

- le code ci-dessous n'est pas à savoir reproduire, mais vous pouvez chercher à le comprendre
- Une autre façon de faire est de choisir 3 axes et de tracer un graphique en 3 dimensions.

In [21]:
```
def extraction_abs_et_ord(indice1,indice2):
    """
    Entrée : deux entiers distinct entre 0 et 3
    Sortie : deux listes de flottants
    Extrait six listes de coordonnées à partir des données de la liste
dataset
    """
    donnees_abscisses_setosa = [iris[indice1] for iris in dataset if iris[4]
== 'Iris-setosa']
    donnees_ordonnees_setosa = [iris[indice2] for iris in dataset if iris[4]
== 'Iris-setosa']
    donnees_abscisses_versicolor = [iris[indice1] for iris in dataset if
iris[4] == 'Iris-versicolor']
    donnees_ordonnees_versicolor = [iris[indice2] for iris in dataset if
iris[4] == 'Iris-versicolor']
    donnees_abscisses_virginica = [iris[indice1] for iris in dataset if
iris[4] == 'Iris-virginica']
    donnees_ordonnees_virginica = [iris[indice2] for iris in dataset if
iris[4] == 'Iris-virginica']
    return
donnees_abscisses_setosa,donnees_ordonnees_setosa,donnees_abscisses_versicolor
donnees_abscisses_virginica,donnees_ordonnees_virginica


donnees_abscisses_setosa,donnees_ordonnees_setosa,donnees_abscisses_versicolor
donnees_abscisses_virginica,donnees_ordonnees_virginica =
extraction_abs_et_ord(0,1)


# Instanciation du graphique
fig, axs = plt.subplots(2, 3, figsize=(20, 10))
# Définition des légendes
point_labels = ["Iris Setosa", "Iris versicolor", "Iris virginica"]
axes_labels = ["longueur des sépales" , "largeur des sépales" , "longueur
des pétales"  , "largeur des pétales"  ]
# Dictionnaire associant la position du graphique avec les caratères étudiés
correspondance = {(0,0):(0,1) , (0,1):(0,2) , (0,2):(0,3) , (1,0):(1,2) ,
(1,1):(1,3) , (1,2):(2,3)}

# Boucle permetant d'afficher les données en choisissant les axes
compteur = 0
for i in range(0,2):
    for j in range(0,3):
        compteur += 1
        # extraction des données
        indice1,indice2=correspondance[(i,j)]

donnees_abscisses_setosa,donnees_ordonnees_setosa,donnees_abscisses_versicolor
donnees_abscisses_virginica,donnees_ordonnees_virginica =
extraction_abs_et_ord(indice1,indice2)
        plt1 =
axs[i,j].plot(donnees_abscisses_setosa,donnees_ordonnees_setosa,
'ro',label='Iris setosa')
```

```
        plt2 =
axs[i,j].plot(donnees_abscisses_versicolor,donnees_ordonnees_versicolor ,
'bv',label='Iris versicolor')
        plt3 =
axs[i,j].plot(donnees_abscisses_virginica,donnees_ordonnees_virginica ,
'gs',label='Iris virginica')
        axs[i,j].set_xlabel(axes_labels[indice1])
        axs[i,j].set_ylabel(axes_labels[indice2])
        axs[i,j].set_title("Représentation des données #"+str(compteur))

fig.legend([plt1,plt2,plt3],     # The line objects
           labels=point_labels,   # The labels for each line
           loc="center right",   # Position of legend
           borderaxespad=0.1,    # Small spacing around legend box
           title="Espèces"  # Title for the legend
           )

plt.show()
```

2.3. Exercice 2

1. Une des espèce vous parraît-elle plus facile à distinguer des autres ? Si oui, laquelle et pourquoi ?
2. La représentation des données #1 est-elle la plus adaptée pour faire des prédictions ? Si non, lesquelles sont plus pertinentes ?
3. Pensez-vous que l'algorithme des plus proches voisins va fonctionner ?

2.3.1. Réponses :

1.
2.
3.

3. Fonctionnement de l'algorithme

Flora se promène dans la nature et trouve deux iris dont elle mesure les pétales et sépales (ceci est réellement arrivé bien sûr !)
Elle trouve les dimensions suivantes :

	largeur des sépales	longueur des sépales	largeur des pétales	longueur des pétales
Iris 1	6	3.7	1.5	0.7
Iris 2	6.5	3.1	5	1.2

Les 2 iris inconnus sont placés dans les représentations graphiques ci-dessous.

In [22]:
```
# Instanciation du graphique
fig, axs = plt.subplots(2, 3, figsize=(20, 10))
# Définition des légendes
point_labels = ["Iris Setosa", "Iris versicolor", "Iris virginica", "Iris
Inconnu 1","Iris Inconnu 2"]
axes_labels = ["longueur des sépales" , "largeur des sépales" , "longueur
des pétales"  , "largeur des pétales"  ]
# Dictionnaire associant la position du graphique avec les caratères
étudiés
correspondance = {(0,0):(0,1) , (0,1):(0,2) , (0,2):(0,3) , (1,0):(1,2) ,
(1,1):(1,3) , (1,2):(2,3)}
iris1= [6,3.7,1.5,0.7,'inconnu 1']
iris2= [6.5, 3.1, 5, 1.2, 'inconnu 2']

# Boucle permetant d'afficher les données en choisissant les axes
compteur = 0
for i in range(0,2):
    for j in range(0,3):
        compteur += 1
        # extraction des données
        indice1,indice2=correspondance[(i,j)]

donnees_abscisses_setosa,donnees_ordonnees_setosa,donnees_abscisses_versicolc
donnees_abscisses_virginica,donnees_ordonnees_virginica =
extraction_abs_et_ord(indice1,indice2)
        plt1 =
axs[i,j].plot(donnees_abscisses_setosa,donnees_ordonnees_setosa,
'ro',label='Iris setosa')
        plt2 =
axs[i,j].plot(donnees_abscisses_versicolor,donnees_ordonnees_versicolor ,
'bo',label='Iris versicolor')
        plt3 =
axs[i,j].plot(donnees_abscisses_virginica,donnees_ordonnees_virginica ,
'go',label='Iris virginica')
        plt4 = axs[i,j].plot(iris1[indice1],iris1[indice2] , marker = '*',
color='black',label='Iris inconnu 1')
        plt5 = axs[i,j].plot(iris2[indice1],iris2[indice2] , marker = '*',
color='m',label='Iris inconnu 2')
        axs[i,j].set_xlabel(axes_labels[indice1])
        axs[i,j].set_ylabel(axes_labels[indice2])
        axs[i,j].set_title("Représentation des données #"+str(compteur))

fig.legend([plt1,plt2,plt3,plt4,plt5],     # The line objects
           labels=point_labels,   # The labels for each line
           loc="center right",   # Position of legend
           borderaxespad=0.1,    # Small spacing around legend box
           title="Espèces"  # Title for the legend
           )

plt.show()
```

3.4. Exercice 3

1. Selon vous, quelles sont les espèces de ces deux iris ? Justifiez votre choix.
2. Expliquer pourquoi la classification de l'iris 2 est plus délicate.

3.4.1. Réponses :

1.
2.

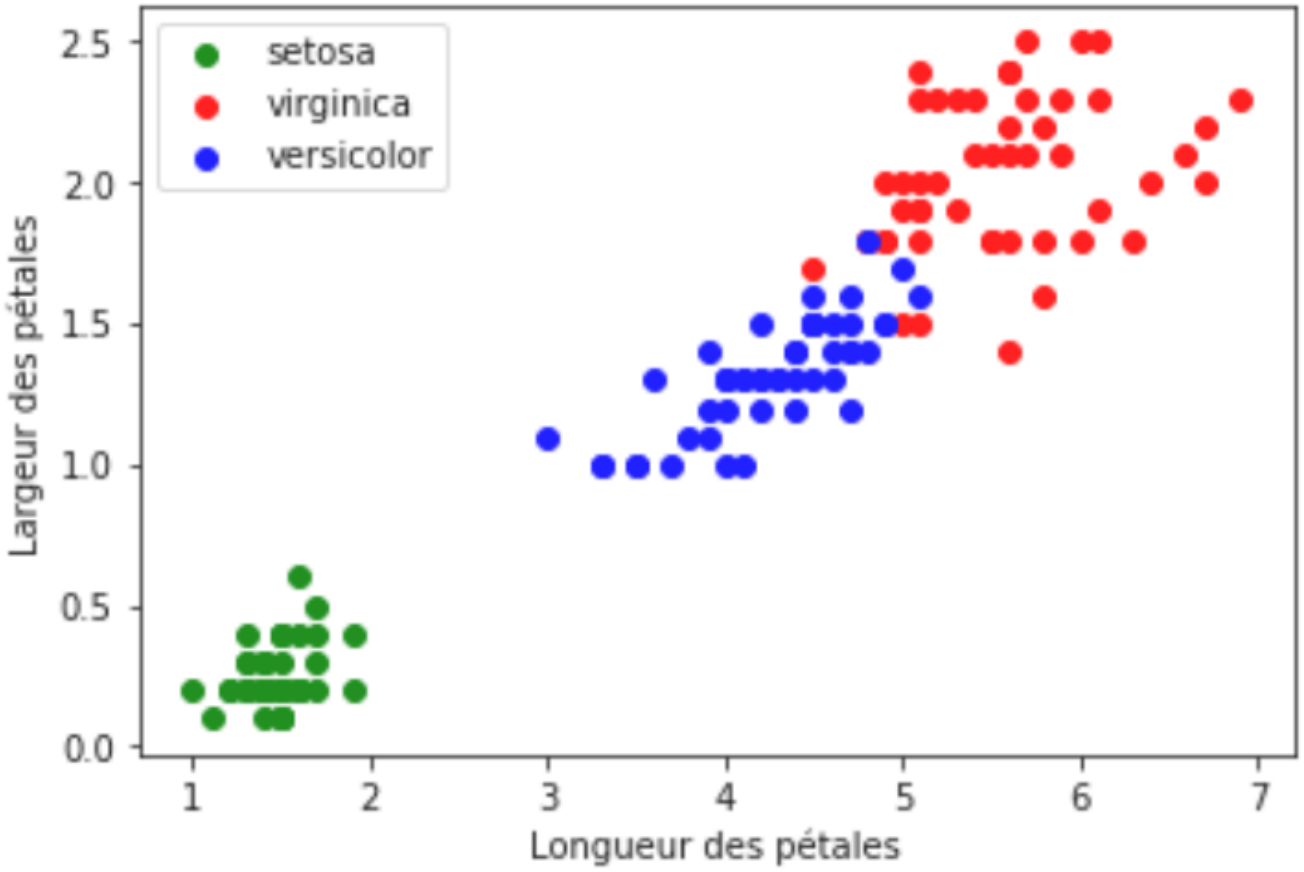
4. Fonctionnement de l'algorithme

Pour avoir une règle plus précise de décision, nous allons utliser l'algorithme des plus proches voisins.

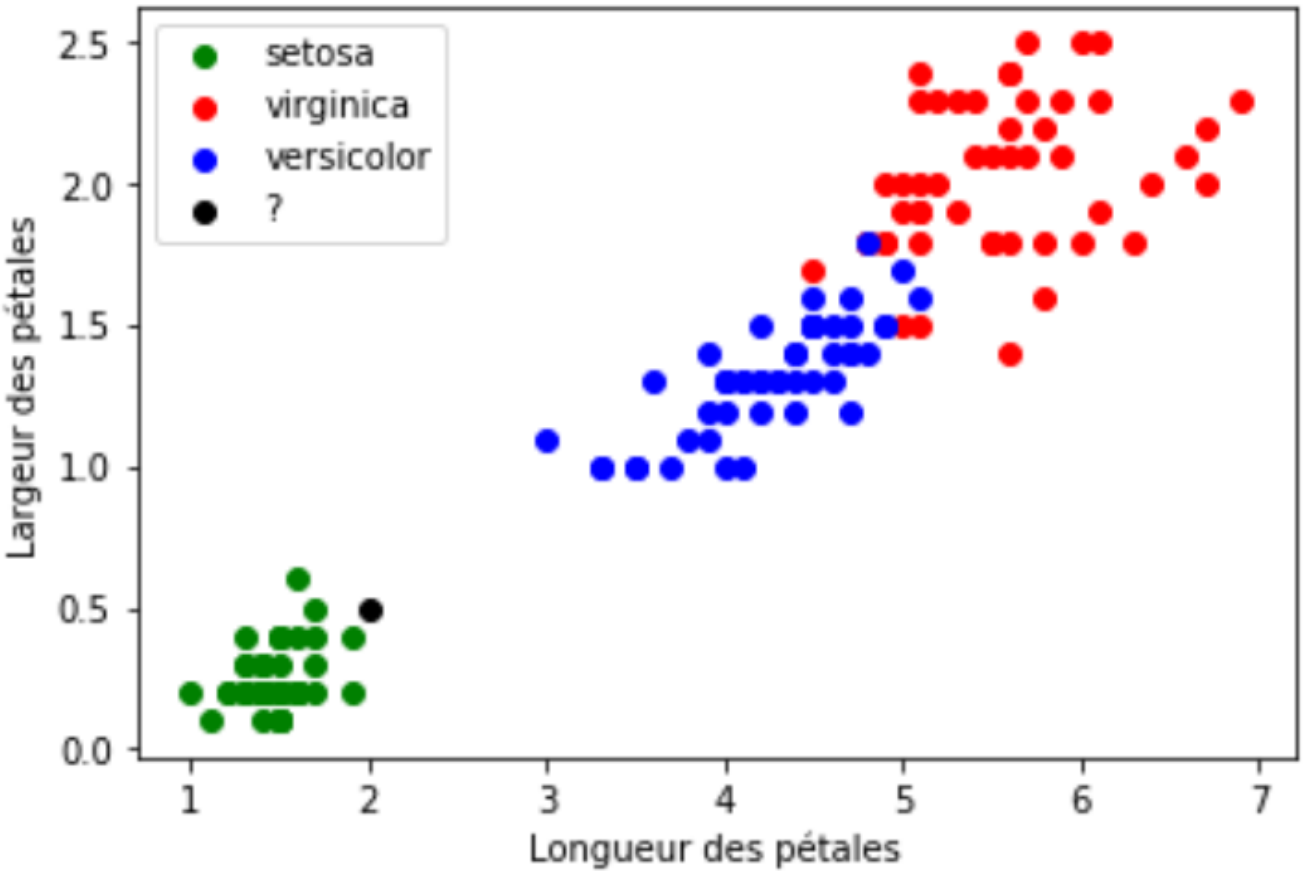
On rappelle le principe de l’algorithme des k plus proches voisins :

1. On calcule les distances entre la donnée et chaque donnée appartenant aux données d’apprentissage.
2. On retient les k données les plus proches de la nouvelle donnée.
3. On attribue à la nouvelle donnée l’étiquette la plus fréquente parmi les k données les plus proches.

Voici une réprésentation en 2D de ce jeu de données :

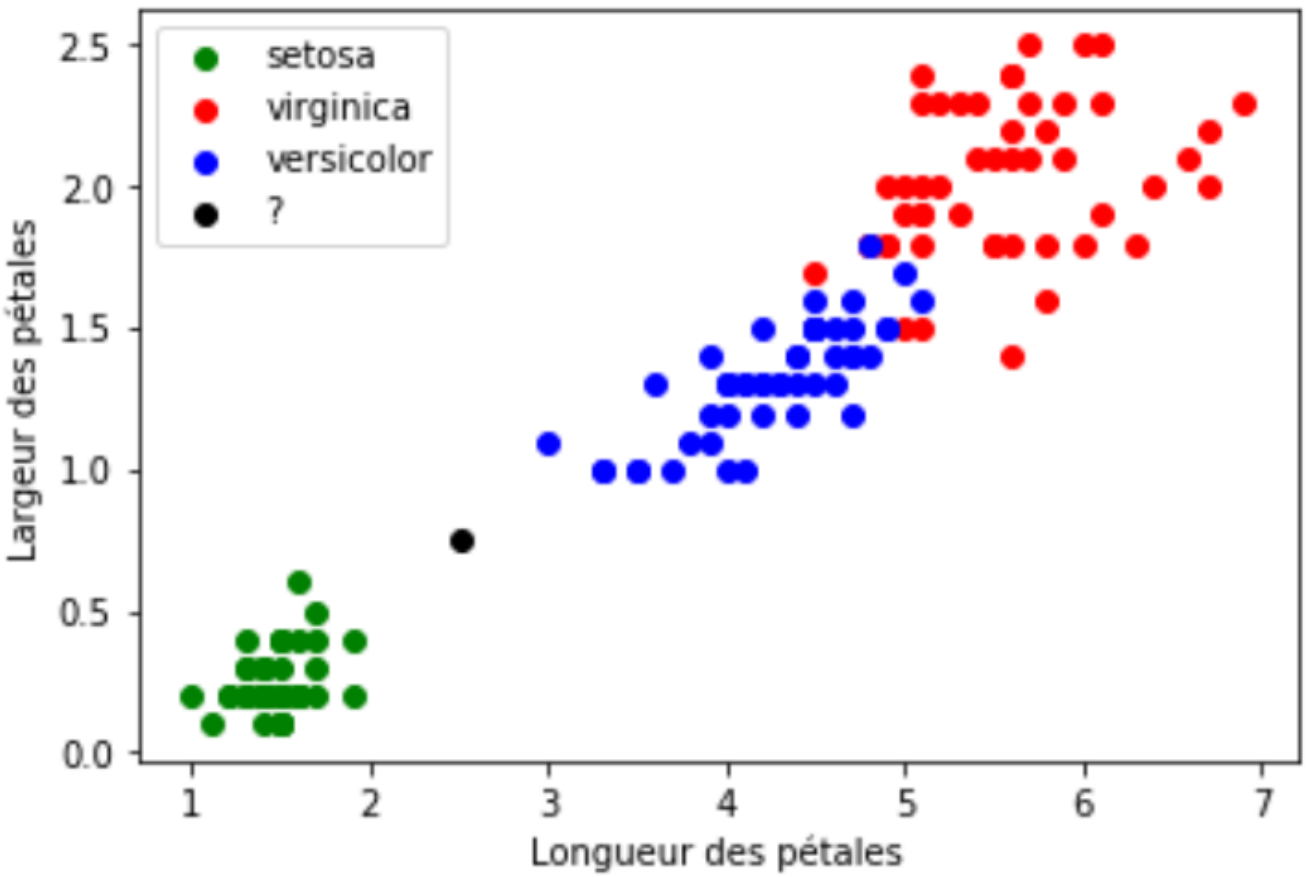


- En abscisse : la longueur des pétales
- En ordonnée : la largeur des pétales



On trouve un nouvel iris dont dont la longueur des pétales est 2 cm et la largeur 0,5 cm.

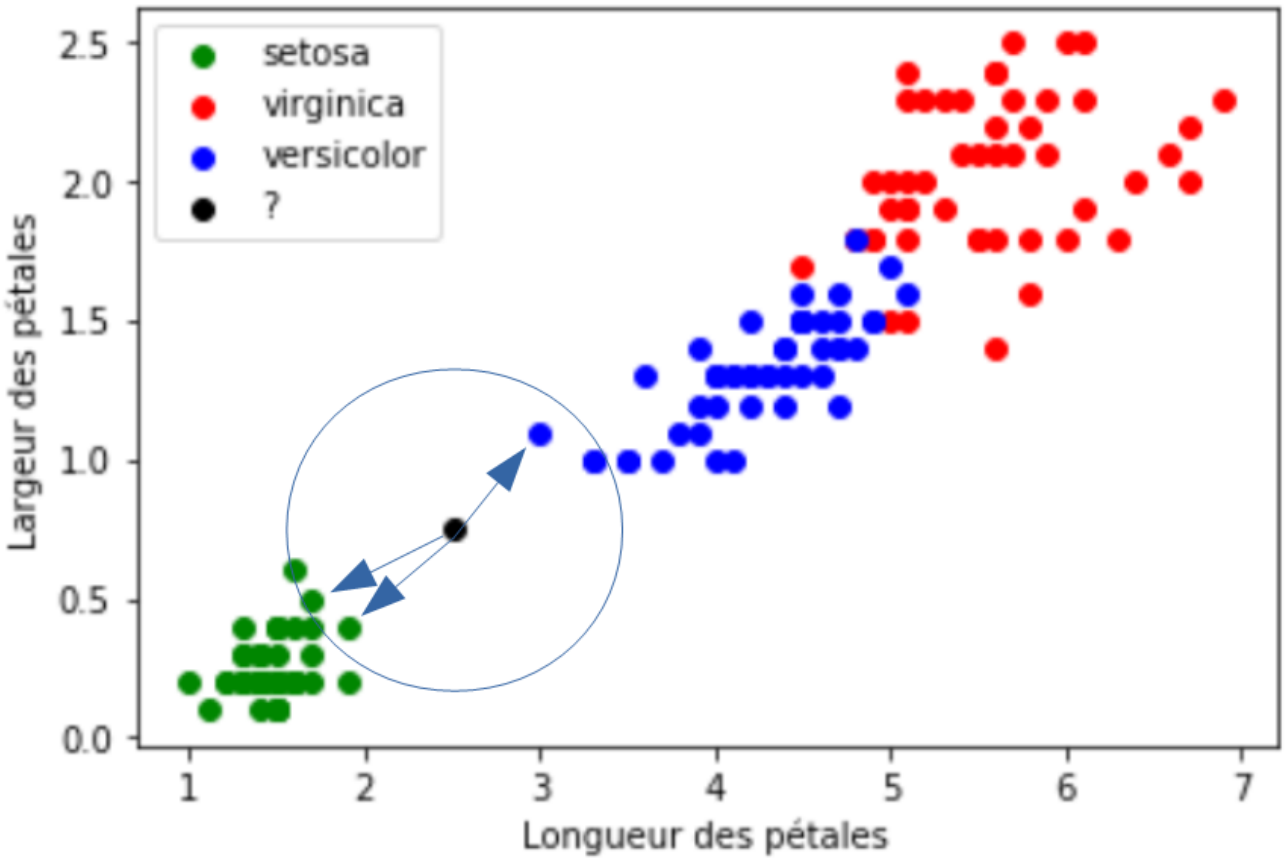
En plaçant le point de coordonnées $(2; 0,5)$, on constate qu'il y a de fortes chances que cet iris soit "iris-setosa"



On trouve un nouvel iris dont dont la longueur des pétales est 2,5 cm et la largeur 0,75 cm.

En plaçant le point de coordonnées $(2,5; 0,75)$, on constate qu'il est plus difficile de prendre une décision.

C'est l'algorithme des k plus proches voisins qui va prendre la décision.



La valeur de k est le nombre de plus proches voisins(en terme de distance) avec lesquels le nouvel iris sera comparé.

Dans l'exemple ci-contre, $k=3$, les trois plus proches voisins sont indiqués par des flèches.

Parmi ces trois voisins, deux sont étiquettés "setosa" et un seul est étiquetté "versicolor".

L'algorithme des 3 plus proches voisins choisira d'étiquetter ce nouvel iris comme "setosa".

4.4.1. Rappel avant de continuer :

- La distance dont il s'agit ici est la distance euclidienne dans un repère orthonormé entre deux points $A(x_{A};y_{A})$ et $B(x_{B};y_{B})$:

$$AB=\sqrt{(x_{B}-x_{A})²+(y_{B}-y_{A})²}$$

- Cette formule ce généralise en dimension supérieure de la façon suivante :

$$AB=\sqrt{(x_{B}-x_{A})²+(y_{B}-y_{A})²+(z_{B}-z_{A})² + ...}$$

5. Programmation de l'algorithme

5.1. Calculer la distance d'un point à chacun des points du dataset.

5.4.1. Exercice 4 :

Compléter la fonction ci-dessous. Elle doit vérifier les containtes suivantes :

- deux paramètres en entrée : deux listes ayant comme paramètres longueur_sepale, largeur_sepale, longueur_petale, largeur_petale comme ceux présents dans table_iris
- renvoie la distance **arrondie à trois décimales** entre ces deux iris en prenant en compte les valeurs de ces quatre champs.

Deux assertions qui doivent être vérifiées par votre fonction sont données ensuite.

Rappels :

- On utilisera `sqrt` du module `math` pour calculer la racine carrée et `**` pour mettre au carré.
- On utilisera `round(x, 3)` pour arrondir un nombre `x` en ne gardant que trois décimales.

```
In [ ]:  def calcul_distance(irisA,irisB):
             """
             Entrée : 2 iris avec 4 variables prédictrices
             Sortie : la distance (euclidienne) entre ces deux iris arrondie à 3
         décimales
             """
             # Votre code ici
             return # Votre code ici
```

```
In [ ]:  # Tests pour vérifier votre fonction calcul_distance
         irisA , irisB =dataset[0] , dataset[1]
         assert calcul_distance(irisA , irisB ) == 0.539
         irisA , irisB =dataset[2] , dataset[3]
         assert calcul_distance(irisA , irisB ) == 0.245
```

5.4.1. Exercice 5 :

Maintenant, pour une nouvelle fleur, il va falloir calculer les distances entre chaque iris du jeu de donnée et la nouvelle fleur.
Pour cela, nous allons écrire une fonction qui ayant :

- en entrée :
  - une liste correspondant à la nouvelle fleur
  - le jeu de données , dans notre exemple c'est `dataset`
- en sortie : une copie du jeu de donnée avec un nouveau champ pour chaque iris correspondant à la distance de cet iris avec la nouvelle fleur.

Remarques :

- Pour copier une liste nommée `liste_originale` dans une liste nommée `liste_copiee`, on peut utliser la commande `liste_copiee = liste_originale[:]`. Ce procédé permet de garder la liste orginale intacte et de procéder à des modification sur la copie.

```
In [ ]:  def distances_dataset(nouvel_iris,datas):
             # Votre code ici
             return # Votre code ici
```

```
In [ ]:  iris1= [6,3.7,1.5,0.7,'inconnu 1']
         iris2= [6.5, 3.1, 5, 1.2, 'inconnu 2']
         distances_iris1 = distances_dataset(iris1,dataset)
         distances_iris2 = distances_dataset(iris2,dataset)
```

La suite va consister à trier la liste obtenue à l'étape précédente par ordre croissant de distance par rapport à la nouvelle fleur.
Ensuite il va falloir extraire les k plus proches voisins.

5.4.1. Exercice 6 :

Pour cela, nous allons écrire une fonction qui ayant :

- en entrée :
  - une entier k (le nombre de plus proches voisins)
  - le jeu de données , dans notre exemple c'est `dataset`, avec le champs distance (obtenu à l'étape précédente)
- en sortie : La liste des étiquettes des k plus proches voisins

Aide : La fonction `sorted` permet de faire un tri 'en place' (la liste initiale est remplacée par la liste triée) en fonction d'un des champs d'une liste.
`sorted(liste, key=lambda liste: liste[2])` permet de trier la liste en fonction du champs d'indice 2.

In [ ]:
```
def extraire_proches_voisins(k,liste_distance):
    # Votre code ici
    return # Votre code ici
```

In [ ]:
```
# Test de la fonction extraire_proches_voisins

assert extraire_proches_voisins(3,distances_iris2) == ['Iris-
versicolor', 'Iris-versicolor', 'Iris-virginica']
assert extraire_proches_voisins(5,distances_iris1) == ['Iris-setosa',
'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa']
```

5.4.1. Exercice 7 :

Ensuite il faut créer une fonction qui permet de déterminer l'étiquette majoritaire dans la liste des k plus proches voisins.

Compléter la fonction `element_majoritaire(etiquettes)` :

- Entrées :
  - La liste `etiquettes`
- Sortie: l'élément majoritaire dans la liste `etiquettes`

In [ ]:
```
def element_majoritaire(etiquettes):
    # elements est le dictionnaire des etiquettes sans doublons. La valeur associée à chaque clé est par défaut 0.
    elements = {element:0 for element in set(etiquettes)}
    # Compléter la ligne suivant pour que la valeur de chaque clé soit le nombre de fois où la clé apparait dans la liste etiquette
    for cle in elements:
        elements[cle] = # Votre code ici
    # Ensuite, à partir du dictionnaire précédent, trouver l'élément majoritaire
    max = 0
    element_majoritaire = ''
    for cle,valeur in elements.items():
        # Votre code ici
    return element_majoritaire
```

In [ ]:
```
#Des tests .
assert
element_majoritaire(['alice','bob','alice','alice','bob','alice','alice
== 'alice'
assert element_majoritaire(['Iris-versicolor', 'Iris-versicolor',
'Iris-virginica']) == 'Iris-versicolor'
```

5.4.1. Exercice 8 :

Intégrez tout ce qui précdède dans une unique fonction `k_plus_proches_voisins` qui prend en paramètres :

- un nombre entier `k`,
- une table d'iris étiquetés `table_iris`,
- et un iris non étiqueté `nouvel_iris`

et renvoie l'étiquette de `iris_a_etiqueter` obtenue selon l'algorithme des k plus proches voisins.

```
In [ ]: def k_plus_proches_voisins(k,nouvel_iris,datas):
            # Votre code ici
            return # Votre code ici
```

5.4.1. Exercice 9 :

En déduire une prédiction pour chacun des deux iris inconnus de l'exercice 4 et vérifier la cohérence avec vos réponses.

```
In [ ]: iris1= [6,3.7,1.5,0.7,'inconnu 1']
        iris2= [6.5, 3.1, 5, 1.2, 'inconnu 2']

        # Votre code ici
```

6. Tester la fiabilité de cet algorithme.

Pour entraîner ou tester un algorithme d'apprentissage automatique, une technique classique est de partager le jeu de données en deux parties :

- un jeu de données d'apprentissage, qui sert à construire le modèle
- un jeu de test qui va servir à tester notre modèle.

Dans le jeu de test, nous connaissons les étiquettes "réelles", nous pouvons donc utiliser notre modèle sur ce jeu de donnée 'de test' pour prédire des étiquettes.
Ensuite, en comparant les étiquettes prédites avec les étiquettes 'réelles', nous pouvons compter les erreurs faites par le modèle pour évaluer celui-ci.
Dans le cas des k plus proches voisins nous pouvons ainsi évaluer le poucentage d'erreurs en fonction de la valeur de k qui est choisie.

6.4.1. Exercice 10 :

Créer une fonction qui permet de séparer le jeu de données en deux jeux apprentissage/test.

Compléter la fonction `creation_jeux` :

- Entrées :
  - le jeux de données : une liste
  - pourcentage_test , un flotant entre 0 et 100 qui représente le % de données consacrées aux tests
- Sortie: deux jeux apprentissage/test - 2 listes

La fonction shuffle du module random peut aider (voir la documentation sur internet)

In [ ]:
```python
# Jeu Apprentissage / Test
#Mélange des données
def creation_jeux(dataset,pourcentage_test):
    # Votre code ici
    return # Votre code ici


jeu_apprentissage,jeu_test = creation_jeux(dataset,20)
```

6.4.1. Exercice 11 :

Créer une fonction qui permet de séparer le jeu de données en deux jeux apprentissage/test.

Compléter la fonction `eval_erreurs` :

- Entrées :
  - le jeux d'apprentissage : une liste
  - le jeux de tests : une liste
  - k un entier supérieur ou égal à 1
- Sortie: le pourcentage d'erreurs réalisés par l'algorithme sur le jeu de test (en se servant du jeu d'apprentissage comme jeu de données)

In [ ]:
```python
def eval_erreurs(jeu_apprentissage,jeu_test,k):
    # Votre code ici
    return # Votre code ici


# Ci-dessous, 100 itérations sont effectués pour calculer la
moyenne du nombre d'erreurs pour chaque valeur de k
# Ensuite un graphique permet de visualiser le pourcentage
d'erreurs en fonction de la valeur de k.
# Comme la génération de la liste `erreurs` est en partie
aléatoire, vos graphiques seront différents à chaque éxécution
de la cellule.
# Mais cela donne une idée de l'influence du paramètre k.
# Attention, l'éxécution de cette cellule peut prendre du temps.

erreurs = []
for k in range(1,12):
    erreur = 0
    for i in range(100):
        erreur += eval_erreurs(jeu_apprentissage,jeu_test,k)
    moyenne = erreur/100
    erreurs.append(moyenne)

# erreurs = [eval_erreurs(jeu_apprentissage,jeu_test,k) for k in
range(1,12)]

print(erreurs)

fig, ax = plt.subplots()  # Create a figure and an axes.
ax.plot(list(range(1,12)), erreurs, label='Erreurs')  # Plot
some data on the axes.
ax.set_xlabel('k')  # Add an x-label to the axes.
ax.set_ylabel("Pourcentage d'erreurs")  # Add a y-label to the
axes.
ax.set_title("Graphique du % d'erreurs en fonction des valeurs
de k")  # Add a title to the axes.
ax.legend()  # Add a legend.
```

7. SUIVANT : Exercice BAC écrit

In [ ]: