Algorithme des k-plus proches voisins

I- INTRODUCTION:

1- Définition et historique :

L'algorithme des k plus proches voisins s'écrit en abrégé k-NN ou KNN , de l'anglais k-nearest neighbors, appartient à la famille des algorithmes d'apprentissage automatique (machine learning). Le terme de machine learning a été utilisé pour la première fois par l'informaticien américain Arthur Samuel en 1959. Les algorithmes d'apprentissage automatique ont connu un fort regain d'intérêt au début des années 2000 notamment grâce à la quantité de données disponibles sur internet.
L'algorithme des k plus proches voisins est un algorithme d'apprentissage supervisé, il est nécessaire d'avoir des données labellisées. À partir d'un ensemble E de données labellisées, il sera possible de classer (déterminer le label) d'une nouvelle donnée (donnée n'appartenant pas à E). À noter qu'il est aussi possible d'utiliser l'algorithme des k plus proches voisins à des fins de régression en statistiques (on cherche à déterminer une valeur à la place d'une classe), mais cet aspect des choses ne sera pas abordé en première.
De nombreuses sociétés (exemple les GAFAM) utilisent les données concernant leurs utilisateurs afin de "nourrir" des algorithmes de machine learning qui permettront à ces sociétés d'en savoir toujours plus sur nous et ainsi de mieux cerné nos "besoins" en termes de consommation.

2- Principe de l'algorithme :

On suppose que l'ensemble ***E*** contiennent ***n*** données labellisées et ***u*** **,** une autre donnée n'appartenant pas à ***E*** qui ne possède pas de label. Soit ***d*** une fonction qui renvoie la distance (qui reste à choisir) entre la donnée *u* et une donnée quelconque appartenant à ***E***. Soit un entier **k** inférieur ou égal à **n**.
Le principe de l'algorithme de k-plus proches voisins est le suivant:

- **On calcule les distances entre la donnée *u* et chaque donnée appartenant à *E* à l'aide de la fonction *d.***
- **On retient les *k* données du jeu de données *E* les plus proches de *u.***
- **On attribue à *u* la classe qui est la plus fréquente parmi les *k* données les plus proches.**

Suivant que l'on raisonne sur une ,deux, trois dimensions, le calcul de la distance entre deux points est plus au moins simple.
Pour appliquer ce principe, il **faudra :**

A.LOUGHANI + JNB

- **Évaluer la distance qui sépare le nouvel élément de chacun des autres points de l'ensemble E. Chaque point de l'ensemble est caractérisé par son indice i.**

- **Stocker ces valeurs de distance d dans une liste du type : [[d,i],[…],...], où d est la distance qui sépare le nouvel élément du point d'indice i.**

- **Trier la liste selon les valeurs des distances d.**

- **Choisir les k premiers points de la liste triée qui sont donc les k-plus proches voisins.**

- **Assigner une classe au nouvel élément en fonction de la majorité des classes représentées parmi les k-plus proches voisins.**

On aura donc besoin de trois fonctions :

- **une fonction** distance **pour calculer la distance entre deux points de coordonnées connues.**

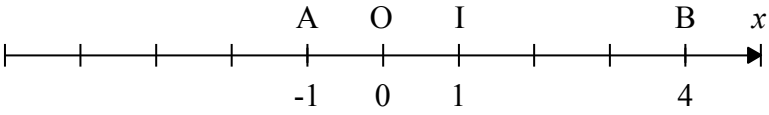
- **Une fonctions** kvoisins **qui détermine les k-plus proches voisins d'un nouvel élément.**

- **Une fonction** predire_classe **qui détermine le résultat majoritaire des classes d'appartenance des k-plus proches voisins et assigne la classe du nouvel élément à cette classe majoritaire.**

II- Notion de distance :

1-Distance Euclidienne et distance de Manhattan:

a-Sur une droite graduée:



distance entre A et B est :

d(A,B) = |abscisse de B – abscisse de A|
= la plus grande abscisse – la plus petite abscisse
= 4-(-1) = 4 + 1 = 5

d(A,B) = 5 unités.

On peut donc écrire une fonction distance, avec Python, qui prend en arguments d'entrée deux coordonnées x1 et x2 et qui renvoie la valeur de la distance entre les deux points de coordonnées x1 et x2

Q1 : tester le programme python ci dessous.

```
1 def distance( x1, x2 ) :
2     return abs( x1- x2 )
```

b- Dans un plan :

On suppose que le plan est muni d'un repère orthonormal (O;I;J).

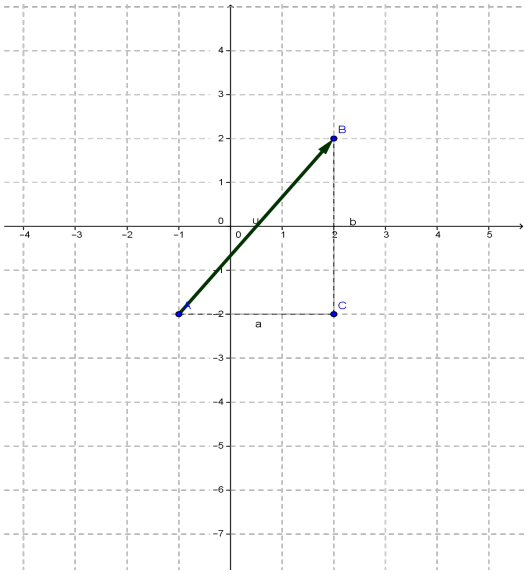
Soient A(x_A;y_A) et B(x_B ;y_B) deux points du plan, la distance euclidienne entre A et B est :

$$\mathbf{d(A,B) = AB =} \sqrt{(xB-xA)^2+(yB-yA)^2}$$

Exemple :

Soit A(-1;-2) et B(2;2) alors
AB= $\sqrt{(2-(-1))^2+(2-(-2))^2}$

$= \sqrt{(2+1)^2+(2+2)^2}$
$= \sqrt{3^2+4^2}$
$= \sqrt{9+16}$
$= \sqrt{25}$
$=5$



AB = 5 unités
En Python , on a :

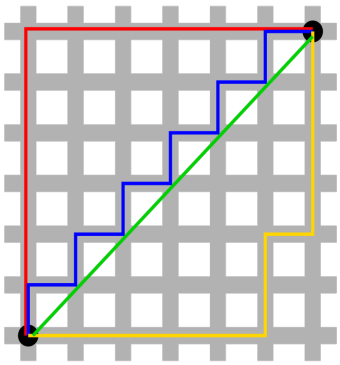
```
from math import sqrt
def distance(p, q):
    """ Renvoie la distance euclidienne entre deux points du plan. """
return sqrt((p[0] - q[0])**2 + (p[1] - q[1])**2)
```

Q2 : Si vous testez ce programme, des erreurs apparaissent : corrigez les pour un fonctionnement correct.
Proposez une amélioration à ce programme.

c- Distance de manhattan :

La distance de Manhattan a été utilisée dans une analyse de régression en 1757 par Roger Joseph Boscovich. L'interprétation géométrique remonte à la fin du XIXe siècle et au développement de géométries non euclidiennes, notamment par Hermann Minkowski et son inégalité de Minkowski, dont cette géométrie constitue un cas particulier, particulièrement utilisée dans la géométrie des nombres (Minkowski 1910).

La distance de Manhattan est appelée aussi taxi-distance, est la distance entre deux points parcourue par un taxi lorsqu'il se déplace dans une ville où les rues sont agencées selon un réseau ou quadrillage. Un taxi-chemin est le trajet fait par un taxi lorsqu'il se déplace d'un nœud du réseau à un autre en utilisant les déplacements horizontaux et verticaux du réseau.



Entre deux points *A* et *B*, de coordonnées respectives (X_A , Y_A) et (X_B , Y_B)la distance de Manhattan est définie par :

$$\mathbf{d\,(A\,,B\,) = |\,X_A - X_B\,| + |\,Y_A - Y_B\,|\,.}$$

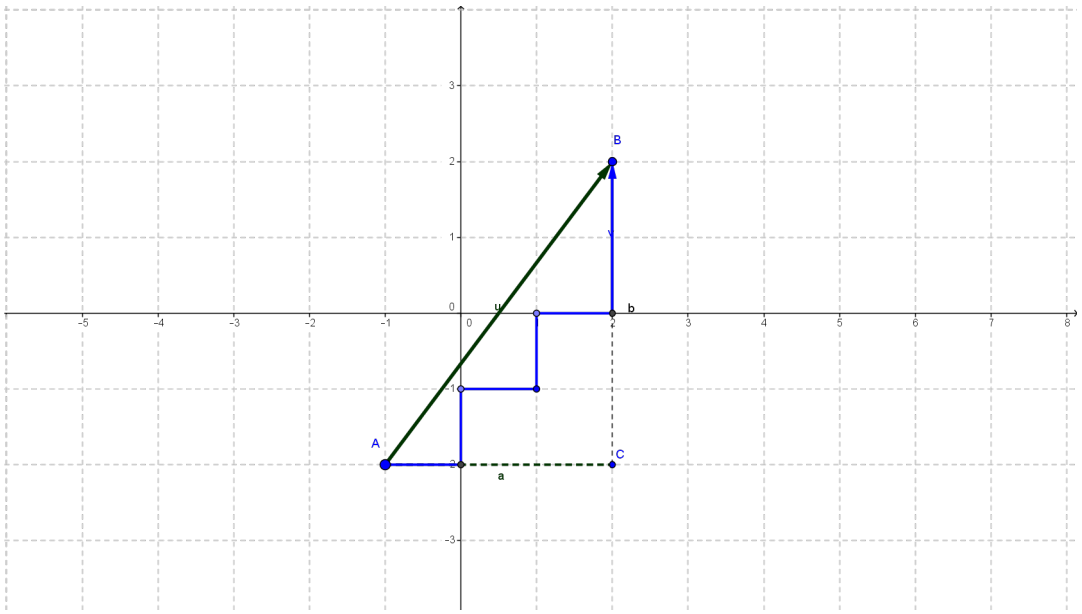
Dans l'exemple ci-dessus, on a :

d(A,B) = | -1-2|+|-2-2|

= |-3|+|-4|

=3+4

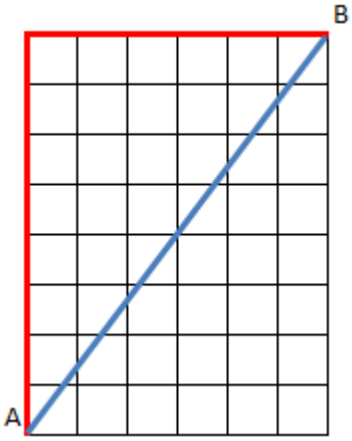
=7



Réservée aux classifications hiérarchiques, la distance de manhattan ne majore pas la pondération des points extrêmes . En revanche, les temps de calcul sont particulièrement longs…

A.LOUGHANI + JNB

Illustration :



Dans le plan ci-dessus, les points A et B sont séparés de 6 unités sur une variable et de 8 unités sur une autre variable. La distance de Manhattan est donc de 14 (en rouge) tandis que la distance euclidienne est la racine carrée de $6^2 + 8^2$, soit 10 (en bleu). Manhattan est donc supérieure de 40 %. Si les points étaient très éloignés, mettons de 20 et 40, Manhattan = 60 et Euclide = 44,7, soit une différence de 34,16 %...

En Python, on a:

```
1 def distmanhattan(xA,yA,xB,yB):
2     distance = abs(xA-xB) + abs(yA-yB)
3     return distance
```

Q3 : tester ce programme

2- Distance de Hamming :

La distance de Hamming est une notion mathématique, définie par Richard Hamming, et utilisée en informatique. Elle permet de quantifier la différence entre deux séquences de symboles. C'est une distance au sens mathématique du terme. À deux suites de symboles de même longueur, elle associe le nombre de positions où les deux suites diffèrent.

Exemples :

Considérons les mots binaires suivants :
a = (0001111) et b = (1101011)
alors d = 1+1+0+0+1+0+0=3
La distance entre *a* et *b* est égale à **3** car 3 bits diffèrent.

- La distance de Hamming entre "**1011101**" et "**1001001**" est 2.
- La distance de Hamming entre "**2143896**" et "**2233796**" est 3.
- La distance de Hamming entre "**ramer**" et "**cases**" est 3.

En Python on a le code suivant pour le calcul de cette distance **de Hamming**:

```
1 def dist_hamming(m1,m2):
2     d = 0
3     for a,b in zip(m1,m2):
4         if a != b :
5             d += 1
6     #return d
7     print(d)
8
9 dist_hamming("ete", "hiver")
```

Q4 : tester ce programme avec les exemples ci dessus puis avec les mots que vous voulez tester

Avec close et cloue, l'instruction dans le console, donne :
>>>dist_hamming("close", "cloue")
>>>d=1.

A	c	l	o	s	e
B	c	l	o	u	e
$d(A,B)$	0	0	0	1	0

```
1  def hamming(c, d) :
2      dist = 0
3      for i in range(min(len(c), len(d))):
4          if c[i]!= d[i] :
5              dist += 1
6          h =  dist + abs(len(c) - len(d))
7          print (h)
8      print()
9      print(dist)
10
11 hamming("longmot", "liongmot")
```

Tester ce programme avec longmot et liongmot ou encore avec longmot et longmoit puis avec les mots que vous voulez tester

3- Distance d'édition :

Les distances d'édition permettent de comparer deux mots entre eux ou plus généralement deux séquences de symboles entre elles. L'usage le plus simple est de trouver, pour un mot mal orthographié, le mot le plus proche dans un dictionnaire, c'est une option proposée dans la plupart des traitements de texte. La distance présentée est la **distance de Levenshtein** . Elle est parfois appelée **Damerau Levenstein Matching (DLM)** . Cette distance fait intervenir trois opérations élémentaires :

- comparaison entre deux caractères
- insertion d'un caractère
- suppression d'un caractère

Pour comparer deux mots, il faut construire une méthode associant ces trois opérations afin que le premier mot se transforme en le second mot.

L'exemple suivant utilise les mots idstzance et distances, il montre une méthode permettant de passer du premier au second. La distance sera la somme des coûts associés à chacune des opérations choisies. La comparaison entre deux lettres identiques est en général de coût nul, toute autre opération étant de coût strictement positif.

mot 1	mot 2	opération	coût
i	d	comparaison entre i et d	1
d	i	comparaison entre d et i	1
s	s	comparaison entre s et s	0
t	t	comparaison entre t et t	0
z		suppression de z	1
a	a	comparaison entre a et a	0

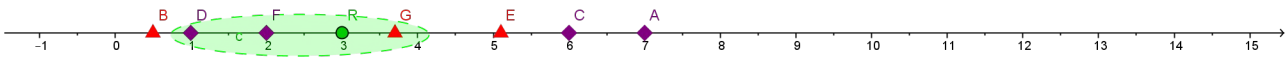
mot 1	mot 2	opération	coût
n	n	comparaison entre n et n	0
c	c	comparaison entre c et c	0
e	e	comparaison entre e et e	0
	s	insertion de s	1
		somme	4

Pour cette distance d'édition entre les mots idstzance et distances. La succession d'opérations proposée n'est pas la seule qui permettent de construire le second mot à partir du premier mais c'est la moins coûteuse.
Avec Python, on a le code suivant pour calculer cette distance :

```python
def distance_edition(mot1, mot2):
    dist = { (-1,-1): 0 }
    for i,c in enumerate(mot1) :
        dist[i,-1] = dist[i-1,-1] + 1
        dist[-1,i] = dist[-1,i-1] + 1
        for j,d in enumerate(mot2) :
            opt = [ ]
            if (i-1,j) in dist :
                x = dist[i-1,j] + 1
                opt.append(x)
                print(opt)
            if (i,j-1) in dist :
                x = dist[i,j-1] + 1
                opt.append(x)
                print(opt)
            if (i-1,j-1) in dist :
                x = dist[i-1,j-1] + (1 if c != d else 0)
                opt.append(x)
                print(opt)
                dist[i,j] = min(opt)
                dis= min(opt)
    #return opt
    print (dis)
mot1 = "idstzance"
mot2 = "distances"
distance_edition(mot1, mot2)
```

III- Applications:

1- Sur une droite graduée :



On cherche les k-plus proches voisins de R(3).
Soit k = 3, si on cherche à lui appliquer un k-NN, ce qui correspond à la partie entourée sur la figure ci-dessus, ses trois plus proches voisins sont G(3.7),F(2)et D(1) qui occupent respectivement la quatrième, la troisième et la deuxième position dans dans la liste, qui sont donc d'indices respectifs, en Python, 3, 2 et 1 car en Python les indices démarrent de 0 :

La liste des positions des éléments(le point R exclus) est :
L = [0.5,1.0,2.0,3.7,5.1,6.0,7.0]
La liste des classe des éléments(le R correspondant au rond vert est exclus) est :
Classes = ['T','C','C','T','T','C','C']# 'C' pour carré et 'T' pour triangle.

Pseudo-code d'une fonction voisin:

n ← nombre d'éléments de L
listeDistanceIndice ← liste vide
pour i allant de 0 à n-1
    d ← distance entre x et L[i]
    ajouter [d,L[i]] en fin de listeDistanceIndice
fin pour
Trier listeDistanceIndice selon d
Voisins ← liste vide
pour i allant de 0 à k-1
  ajouter listeDistanceIndice[i][1] à la fin de Voisins
fin pour
renvoyer Voisins

Fonction Kvoisins en Python:

```python
def distance( a, b ) :
    return abs( a - b )

def Kvoisins(L,k,x) :
    listeDistanceIndice = []
    for i in range(len(L)) :
        d = distance(x,L[i])
        listeDistanceIndice.append([d,i])
    listeDistanceIndice.sort()
    Voisins = []
    for i in range(k):
        Voisins.append(listeDistanceIndice[i][1])
    return Voisins

L = [0.5,1.0,2.0,3.7,5.1,6.0,7.0]
Classes = ['T','C','C','T','T','C','C']
x = 3.0
k = 3
print("liste des indices",Kvoisins(L,k,x))
print("liste des voisins",[L[i] for i in Kvoisins(L,k,x)])
```

Ce qui donne dans le shell :
liste d'indices[3, 2, 1]
liste des voisins [3.7, 2.0, 1.0]

Classification (Attribution de classes) :

On souhaite attribuer une nouvelle classe à l'élément R(rond vert) qui tienne compte de la classe

moyenne de ses plus proches voisins.Dans l'exemple en cours, k-NN avec k = 3, dans la liste des voisins ,il y a plus d'éléments de classes 'C' que d'éléments de classe 'T', donc on attribue la classe 'C ' à R(Rond vert).
Si k =5 , par exemple, il y aurait trois éléments de classe 'T' et deux de classes 'C'. Dans ce cas on attribuerait la classe 'T' à l'élément R.

Pseudo-code de classification :

Voisins ← Kvoisins(L,k,x)
Classespossibles ← ['C', 'T']
decompte ← [0, 0]
pour v allant de début à la fin de Voisins
  si Classe[v] est un carré
    ajouter 1 au 1er élément de decompte
  sinon
    ajouter 1 au 2-ième élément de decompte
  fin si
fin pour
indice ← indice du maximum de decompte
renvoyer Classespossibles [indice]

Fonction predire_classe en Python :

```
22 def predire_classe(L,Classes,k,x) :
23     Voisins = Kvoisins(L,k,x)
24     Classespossibles = ['C', 'T']
25     decompte = [0, 0]
26     for v in Voisins:
27         if Classes[v] == 'C' :
28             decompte[0]+= 1
29         else :
30             decompte[1]+=1
31     plusGrandDecompte = decompte[0]
32     indice = 0
33     if decompte[1] > plusGrandDecompte :
34         indice = 1
35     return Classespossibles[indice]
36 print("Classe du nouvel élément :",predire_classe(L,Classes,k,x))
```

2-Dans le plan :

Nous allons utiliser le jeu de données "iris" relativement connu dans le monde du machine learning .

En 1936, Edgar Anderson a collecté des données sur 3 espèces d'iris : "iris setosa", "iris virginica" et "iris versicolor"

iris setosa

iris virginica

iris versicolor

Pour chaque iris étudié, Anderson a mesuré (en cm) :

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

Par souci de simplification, nous nous intéresserons uniquement à la largeur et à la longueur des pétales.

Pour chaque iris mesuré, Anderson a aussi noté l'espèce ("iris setosa", "iris virginica" ou "iris versicolor")

Vous trouverez 50 de ces mesures dans le fichier iris.csv .

En résumé, vous trouverez dans ce fichier :

- la longueur des pétales
- la largeur des pétales
- l'espèce de l'iris (au lieu d'utiliser les noms des espèces, on utilisera des chiffres : 0 pour "iris setosa", 1 pour "iris virginica" et 2 pour "iris versicolor")

	A	B	C
1	petal_length	petal_width	species
2	1.4	0.2	0
3	1.4	0.2	0
4	1.3	0.2	0
5	1.5	0.2	0
6	1.4	0.2	0
7	1.7	0.4	0
8	1.4	0.3	0
9	1.5	0.2	0
10	1.4	0.2	0
11	1.5	0.1	0

extrait du jeu de données "iris"

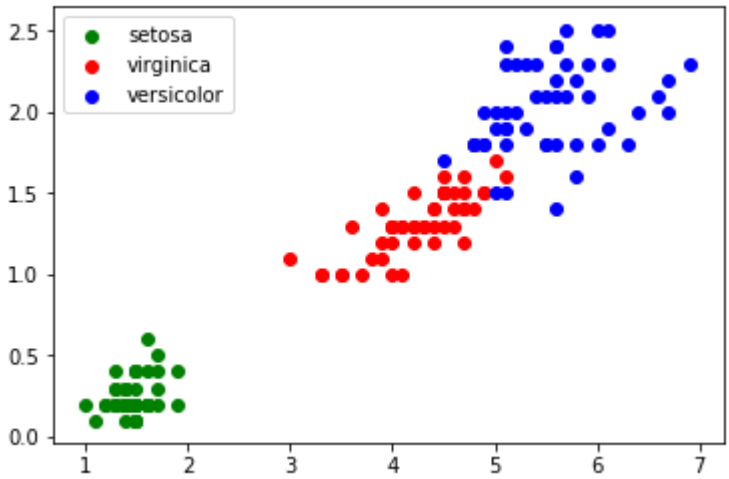
a- Placer le fichier iris.csv dans le même répertoire que votre fichier Python, étudiez et testez le code suivant :

```
import pandas
import matplotlib.pyplot as plt
iris=pandas.read_csv("iris.csv")
x=iris.loc[:,"petal_length"]
y=iris.loc[:,"petal_width"]
lab=iris.loc[:,"species"]
plt.scatter(x[lab == 0], y[lab == 0], color='g', label='setosa')
plt.scatter(x[lab == 1], y[lab == 1], color='r', label='virginica')
plt.scatter(x[lab == 2], y[lab == 2], color='b', label='versicolor')
plt.legend()
plt.show()
```

Quelques explications sur le programme ci-dessus :

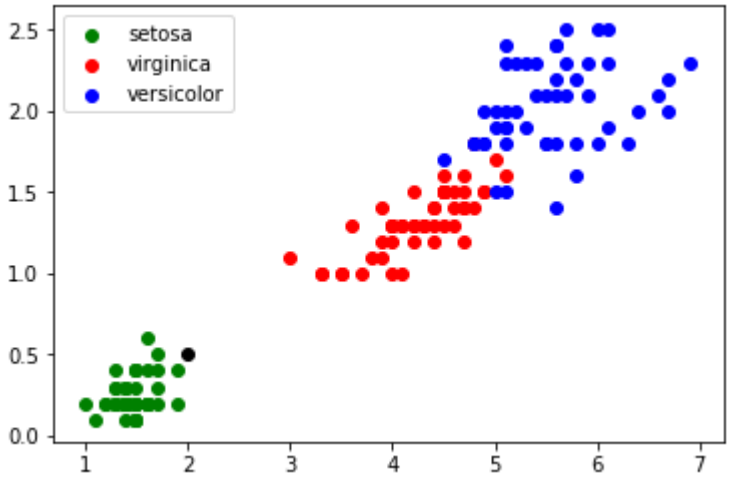
- La partie "Pandas" ne devrait pas vous poser de problèmes : x correspond à la longueur des pétales, correspond à la largeur des pétales et lab correspond à l'espèce d'iris (0,1 ou 2)
- Nous utilisons ensuite la bibliothèque matplotlib qui permet de tracer des graphiques très facilement. "plt.scatter" permet de tracer des points, le "x[lab == 0]" permet de considérer uniquement l'espèce "iris setosa" (lab==0). Le premier "plt.scatter" permet de tracer les points correspondant à l'espèce "iris setosa", ces points seront vert (color='g'), le deuxième "plt.scatter" permet de tracer les points correspondant à l'espèce "iris virginica", ces points seront rouge (color='r'), enfin le troisième "plt.scatter" permet de tracer les points correspondant à l'espèce "iris versicolor", ces points seront bleu (color='b'). Nous aurons en abscisse la longueur du pétale et en ordonnée la largeur du pétale.

On obtient :



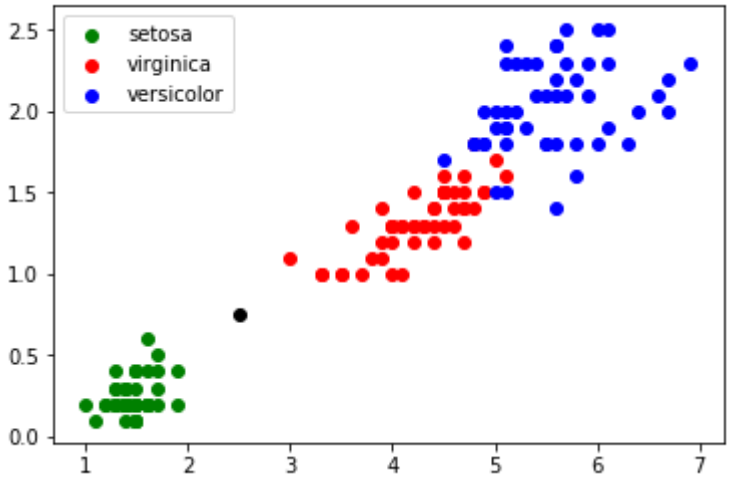
Nous obtenons des "nuages" de points, on remarque ces points sont regroupés par espèces d'iris (sauf pour "iris virginica" et "iris versicolor", les points ont un peu tendance à se mélanger).

Imaginez maintenant qu'au cours d'une promenade vous trouviez un iris, n'étant pas un spécialiste, il ne vous est pas vraiment possible de déterminer l'espèce. En revanche, vous êtes capables de mesurer la longueur et la largeur des pétales de cet iris. Partons du principe qu'un pétale fasse 0,5 cm de large et 2 cm de long. Plaçons cette nouvelle donnée sur notre graphique (il nous suffit d'ajouter la ligne "plt.scatter(2.0, 0.5, color='k')", le nouveau point va apparaitre en noir (color='k')) :



Remarque : Il y a de fortes chances que votre iris soit de l'espèce "iris setosa".

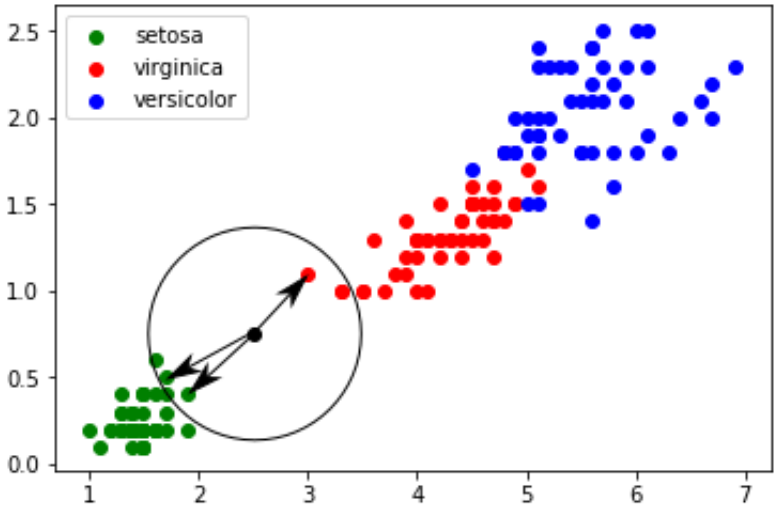
Il est possible de rencontrer des cas plus difficiles, par exemple : largeur du pétale = 0,75 cm ; longueur du pétale = 2,5 cm :



Dans ce genre de cas, il peut être intéressant d'utiliser l'algorithme des "k plus proches voisins", en quoi consiste cet algorithme :

- on calcule la distance entre notre point (largeur du pétale = 0,75 cm ; longueur du pétale = 2,5 cm) et chaque point issu du jeu de données "iris" (à chaque fois c'est un calcul de distance entre 2 points tout ce qu'il y a de plus classique)
- on sélectionne uniquement les k distances les plus petites (les k plus proches voisins)
- parmi les k plus proches voisins, on détermine quelle est l'espèce majoritaire. On associe à notre "iris mystère" cette "espèce majoritaire parmi les k plus proches voisins"

b- Prenons k = 3



Les 3 plus proches voisins sont signalés ci-dessus avec des flèches : nous avons deux "iris setosa" (point vert) et un "iris virginica" (point rouge). D'après l'algorithme des "k plus proches voisins", notre "iris mystère" appartient à l'espèce "setosa".

La bibliothèque Python Scikit Learn propose un grand nombre d'algorithmes lié au machine learning (c'est sans aucun doute la bibliothèque la plus utilisée en machine learning). Parmi tous ces algorithmes, Scikit Learn propose l'algorithme des k plus proches voisins.

c- Après avoir placé le fichier iris.csv dans le même répertoire que votre fichier Python, étudiez et testez le code suivant :

```
import pandas
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
#traitement CSV
iris=pandas.read_csv("iris.csv")
x=iris.loc[:,"petal_length"]
y=iris.loc[:,"petal_width"]
lab=iris.loc[:,"species"]
#fin traitement CSV
#valeurs
longueur=2.5
largeur=0.75
k=3
#fin valeurs
#graphique
plt.scatter(x[lab == 0], y[lab == 0], color='g', label='setosa')
plt.scatter(x[lab == 1], y[lab == 1], color='r', label='virginica')
plt.scatter(x[lab == 2], y[lab == 2], color='b', label='versicolor')
plt.scatter(longueur, largeur, color='k')
plt.legend()
#fin graphique
#algo knn
d=list(zip(x,y))
model = KNeighborsClassifier(n_neighbors=k)
model.fit(d,lab)
prediction= model.predict([[longueur,largeur]])
#fin algo knn
#Affichage résultats
txt="Résultat : "
if prediction[0]==0:
    txt=txt+"setosa"
if prediction[0]==1:
    txt=txt+"virginica"
if prediction[0]==2:
    txt=txt+"versicolor"
plt.text(3,0.5, f"largeur : {largeur} cm longueur : {longueur} cm", fontsize=12)
plt.text(3,0.3, f"k : {k}", fontsize=12)
plt.text(3,0.1, txt, fontsize=12)
#fin affichage résultats
plt.show()
```

Le programme ci-dessus n'est pas très complexe à comprendre, nous allons tout de même nous attarder sur la partie "knn" :

```
d=list(zip(x,y))
model = KNeighborsClassifier(n_neighbors=k)
model.fit(d,lab)
prediction= model.predict([[longueur,largeur]])
```

La première ligne "d=list(zip(x,y))" permet de passer des 2 listes x et y :

```
x = [1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, ...]
y = [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.4,....]
```

à une liste de tuples d :

```
d = [(1.4, 0.2), (1.4, 0.2), (1.3, 0.2), (1.5, 0.2), (1.4, 0.2), (1.7, 0.2), (1.4, 0.4), ...]
```

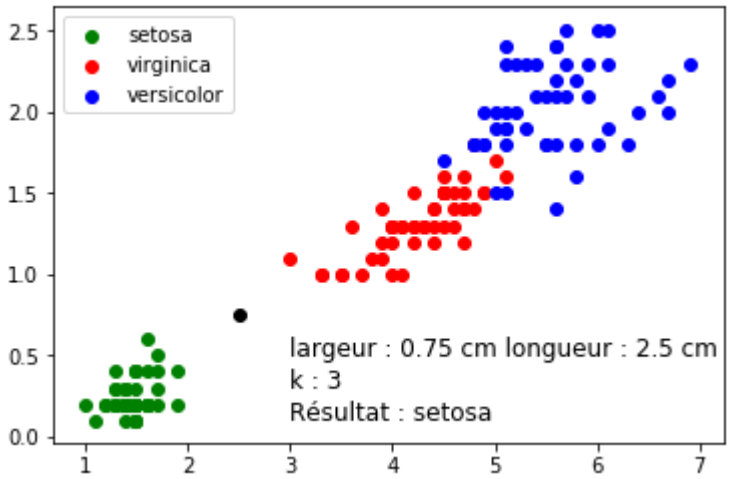
les éléments des tableaux x et y ayant le même indice sont regroupés dans un tuple, nous obtenons bien une liste de tuples.

"KNeighborsClassifier" est une méthode issue de la bibliothèque scikit-learn (voir plus haut le "from sklearn.neighbors import KNeighborsClassifier"), cette méthode prend ici en paramètre le nombre de "plus proches voisins" (model = KNeighborsClassifier(n_neighbors=k))

"model.fit(d, lab)" permet d'associer les tuples présents dans la liste "d" avec les labels (0 : "iris setosa", 1 : "iris virginica" ou 2 : "iris versicolor"). Par exemple le premier tuple de la liste "d", (1.4, 0.2) est associé au premier label de la liste lab (0), et ainsi de suite...

La ligne "prediction= model.predict([[longueur,largeur]])" permet d'effectuer une prédiction pour un couple [longueur, largeur] (dans l'exemple ci-dessus "longueur=2.5" et "largeur=0.75"). La variable "prediction" contient alors le label trouvé par l'algorithme knn. Attention, "predection" est une liste Python qui contient un seul élément (le label), il est donc nécessaire d'écrire "predection[0]" afin d'obtenir le label.

On a :



d- Modifiez le programme du "À faire vous-même 2" afin de tester l'algorithme knn avec un nombre "de plus proches voisins" différent (en gardant un iris ayant une longueur de pétale égale à 2,5 cm et une largeur de pétale égale à 0,75 cm). Que se passe-t-il pour k = 5 ?

e- Testez l'algorithme knn (toujours à l'aide du programme du "À faire vous-même 2") avec un iris de votre choix (si vous ne trouvez pas d'iris, vous pouvez toujours inventer des valeurs ;-)) .

3- Recherche du mot le plus proche d'un lexique :

On parcourt un tableau non vide lexique contenant des mots en enregistrant le mot le plus proche vu jusque là dans une variable mot_dist_min. Cette variable est initialisée avec le premier mot du lexique qu'on a supposé non vide.

On utilise ici la distance de Hamming:

```
def dist_hamming(m1,m2):
    dm = 0
    for a,b in zip(m1,m2):
        if a != b :
            dm += 1
    return dm

lexique=["camion", "action", "bastion", "question","canon", "fanion", "caution" ]

m = input("Entrer un mot : ")
dist_min = dist_hamming(lexique[0], m)
mot_dist_min = lexique[0]

print(mot_dist_min)
for l in lexique :
    d = dist_hamming(l, m)
    print(d)
    if  d< dist_min :
        mot_dist_min = l
        dist_min = d

print(mot_dist_min)
```

Références :

-Cécile Canu, Numériques et sciences de l'informatique , Travailler en autonomie, 1re Nouveaux programmes, ellipses.
-Thibaut Balabonski, Numériques et sciences de l'informatique , 30 leçon avec exercices corrigés, ellipses
-edscole education.fr
- David Roche, pixees.fr
-matlasups.fr
-xavierdupre.fr
-Wikiedia.org
-pedagogie.ac-nantes.fr
-101computing.net
-python-course.eu
-iut.info.univ-reims.fr.
-jybaudot.fr