Chapitre 13. Algorithme des k plus proches voisins (knn)

Table des matières

Remplissez le jupyter notebook suivant en vous aidant de votre livre de Première NSI de Serge BAYS .

- Pour répondre, double-cliquez sur **Réponse** et complétez la zone en-dessous. Puis cliquez sur le bouton *Exécuter*.
- **Important : pour fermer votre jupyter notebook, cliquez sur :**

Fichier / Créer une nouvelle sauvegarde

puis sur :

Fichier / Fermer et Arrêter

- Ecrivez ci-dessous votre prénom et votre nom :

Réponse :

Chapitre 13. Algorithme des k plus proches voisins

1. Introduction

L'algorithme des k plus proches voisins "Nearest Neighbors" (algorithme kNN)

A la base de l'intelligence artificielle, il y a l'apprentissage.

Parmi les types d'apprentissage, il y a l'apprentissage supervisé.

Parmi les méthodes d'apprentissage supervisé, il y a l'algorithme des k plus proches voisins.

Principe de l'apprentissage supervisé avec l'algorithme des k plus proches voisins sur un exemple :

- On entre dans la machine un ensemble de données de référence.

Par exemple nous connaissons $n = 10$ individus A, B, C, D, E, F, G, H, I, J et pour chacun on connait :

- Sa caractéristique ("feature"). Ici dans l'exemple, c'est $x' = nombre\,d'amis\,sur\,Facebook$
- Son étiquette ("label") $y'\ aime$ ou $n'aime\,pas$ un film donné.

Pour plus de facilité, on attribué la couleur bleue à ceux qui aiment le film et la couleur rouge à ceux qui ne l'aiment pas.

Voici la table des données de référence :

Individu	Caractéristique (nombre d'amis) x'	Etiquette (aime ou non le film) y'
A	1000	aime
B	200	n'aime pas
C	300	n'aime pas
D	350	aime
E	800	n'aime pas
F	80	n'aime pas
G	400	aime
H	**100**	n'aime pas
I	250	n'aime pas
J	780	n'aime pas
P	x = 92	y = ?

Il y a un onzième individe "P" dont on ne connait que la "feature" : il a 92 amis Facbook.

On veut prédire, à partir de sa caractéristique x = 92, en se basant sur l'apprentissage qui a été fait avant (on a entré les 10 points de A à J), la catégorie à laquelle appartiendra P (soit on le classe parmi ceux qui aiment, soit parmi ceux qui n'aiment pas le film).

ceux qui aiment, soit parmi ceux qui n'aiment pas le film).

- On cherche, dans l'ensemble dix individus (de A à J), lesquels sont **les k plus proches voisins** d'un onzième individu P. Disons pour notre exemple, que nous traaillons avec k = 3.

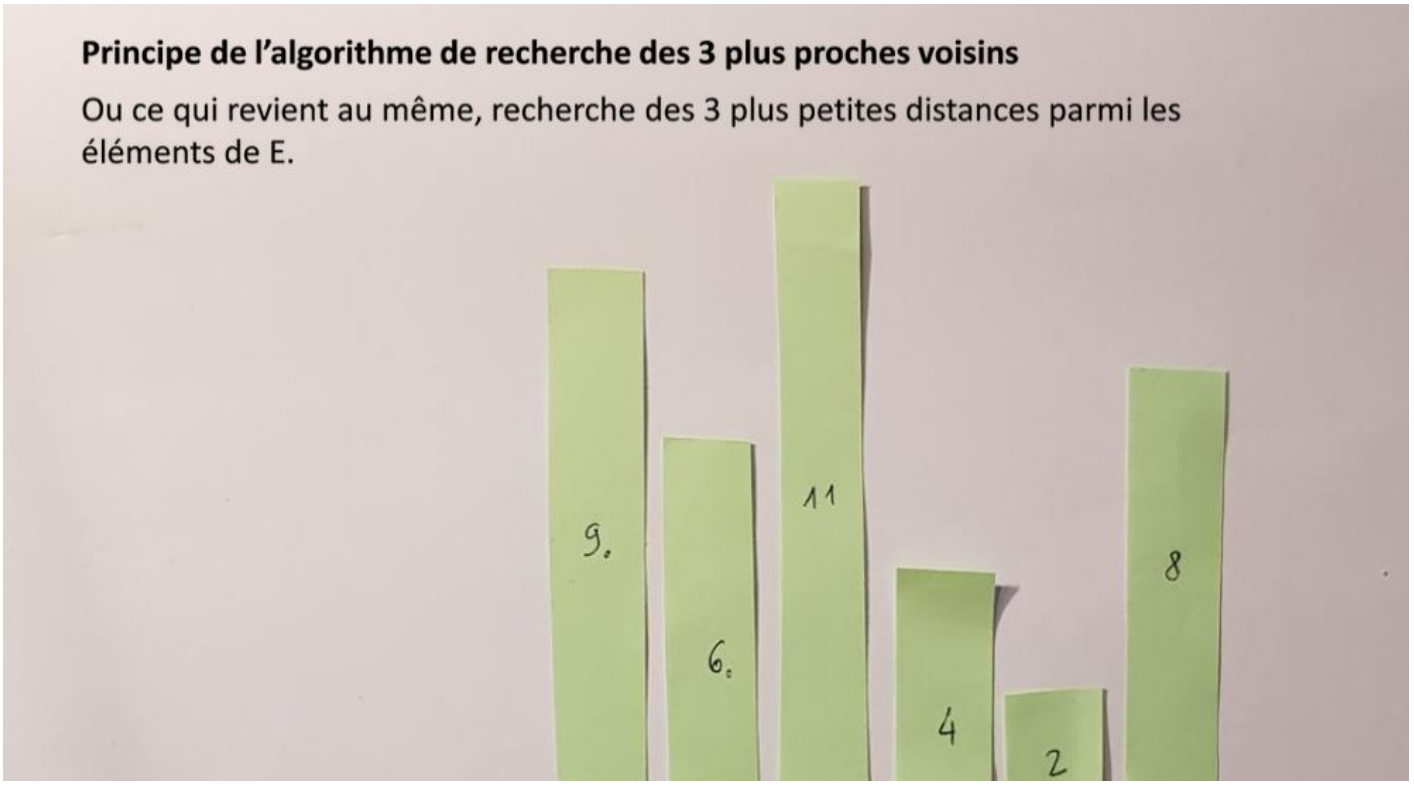
- Donc on fera dans l'ordre :
  1. La liste des distances entre P et chacun des 10 points.
  1. Extraire la sous-liste des 3 plus petites distances (autrement dit la liste des trois plus proches voisins de P).
  1. Regarder les trois étiquettes des trois plus proches voisins. On pourra alors prédire que l'individu P aura la même étiquette que l'étiquette majoritaire parmi les trois.

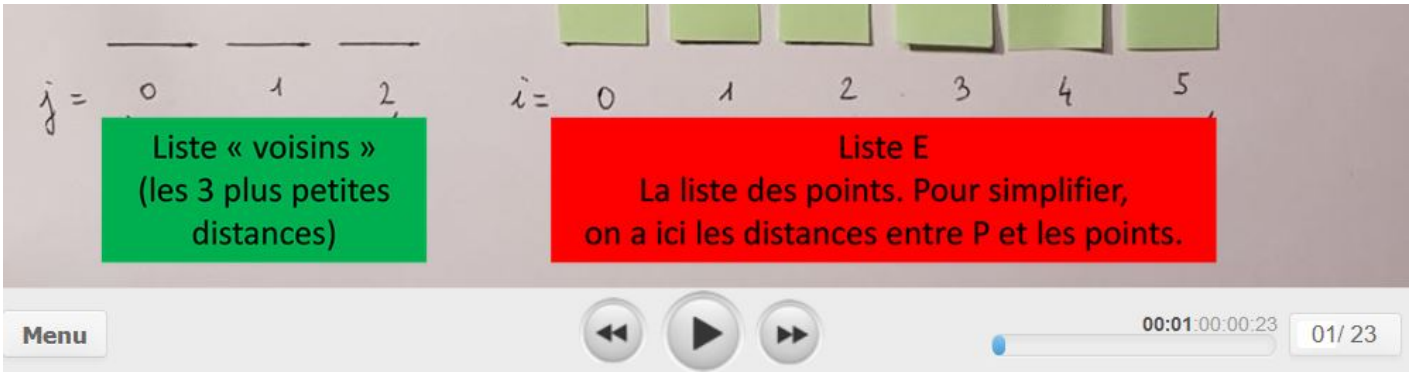
1. et 2. forment la phase d'apprentissage supervisé c'est à dire l'apprentissage à partir d'exemples connus ou "expertisés".
2. est la phase de prédiction de l'étiquette pour un nouvel individu P extérieur au groupe des 10 premier individus.

2. Recherche des k plus proches voisins sur un tableau de dimension 1

Le principe de l'algorithme avec 6 points.

- Cliquez sur l'image ci-dessous pour démarrer le diaporama de présentation du **principe de la recherche des 3 plus petites valeurs dans une liste** (qui correspondent aux 3 plus petites distances entre les 6 points connus et un septième point P extérieur au groupe des 6).





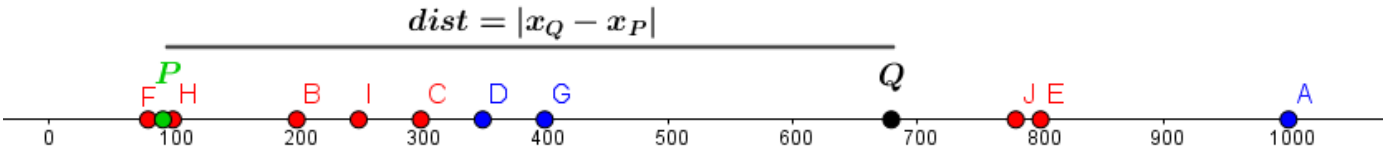
(http://www.astrovirtuel.fr/jupyter/19_pnsi_cours/principe_k_plus_proches_voisins/principe_k_plus_proche

L'algorithme lui-même avec 10 points.

Illustration de l'exemple de la recherche des k plus proches voisins de *P* en "dimension 1" c'est à dire avec 1 coordonnée.

On a remplacé les étiquettes par un code couleur : Rouge pour "n'aime pas le film", Bleu pour "aime le film".

Vert est la couleur de l'individu extérieur aux données, dont on cherche à prédire s'il aime ou n'aime pas le film.



Données :
10 points de A à J

- A = (1000)
- B = (200)
- C = (300)
- D = (350)
- E = (800)
- F = (80)
- G = (400)
- H = (100)
- I = (250)
- J = (780)
- P = (92)

En bleu : aime
En rouge: n'aime pas

2.1 Explication de l'algorithme des k plus proches voisins avec un diaporama.

- Cliquez sur l'image ci-dessous pour démarrer le diaporama.
  - Il y a un premier diaporama rapide de présentation générale de l'algorithme.
  - Puis il y a un second diaporama beaucoup plus long qui déroule complètement le traitement de l'exemple ci-dessus.

	i			i						
	0	1	2	3	4	5	6	7	8	9
E =	A	B	C	D	E	F	G	H	I	J
Distance à P	908	108	208	258	708	12	308	8	158	688

	j		
	0	1	2
voisins =			
Distance à P			

Algorithme des k plus proches voisins

```
voisins = []
for i in range(k):
    voisins.append(E[i])

for i in range(k, len(E)):
    dist = d(P, E[i])
    u = i

    for j in range(k):
        if  dist < d(voisins[j], P):
            dist = d(voisins[j], P)
            u = j

    if u != i:
        voisins[u] = E[i]
return voisins
```

(http://www.astrovirtuel.fr/jupyter/19_pnsi_cours/animation_k_plus_proches_voisins_version1/animation_k

2.2 Explication de l'algorithme des k plus proches voisins avec des commentaires.

2.2.1 Première version avec les distances directement dans la liste E.

In [ ]:
```
# Dans un premier temps, pour simplifier, on suppose qu'on a deja cal
cule les distances
# entre P et les 10 points A, B, ..., J.
# Donc E = [908, 108, 208, 258, 708, 12, 308, 8, 158, 688].

# On cherche les k = 3 plus proches voisins de P, c'est à dire les 3
plus petits E[i].
# A la fin du programme la liste voisins contiendra les 3 plus petits
elements E[i].



E = [908, 108, 208, 258, 708, 12, 308, 8, 158, 688]
voisins = []

# Initialisation de la liste voisins avec les 3 premiers elements de
E.
for i in range(3):
```

```
for i in range(3):
    voisins.append(E[i])

for i in range(3, 10):
    dist = E[i] # Initialisation de dist =
    u = i  # On stocke dans u l'indice i correspondant a dist

    # Recherche du max (E[i], voisins[0], voisins[1], voisins[2]).
    for j in range(3):
        if voisins[j] > dist:
            dist = voisins[j]
            u = j  # On stocke dans u l'indice j correspondant a dist

    if u != i: # Si l'indice du max correspond a un element de voisin
s
        voisins[u] = E[i] # alors on remplace cet element par E[i]

print("E = ", E)
print("voisins = ", voisins)
```

A retenir :

Cet algorithme des *k plus proches voisins* contient trois boucles for :

- Première boucle for: Pour remplir au départ la liste *voisins* avec les 3 premiers éléments E[0], E[1], E[2].
- Deuxième boucle for: Elle contient trois parties :
  - La distance entre P et le point E[i] est stockée dans **dist**. avec stockage dans u de l'indice i de l'élément.
  - Une troisième boucle for: Elle contient un premier if: c'est l'algorithme de recherche du maximum de (dist, voisins[0], voisins[1], voisins[2]) avec stockage dans u de l'indice j correspondant au point qui donne la distance maximale et qui a tét stockée dans **dist**.
  - Un deuxième if: Si le point qui donne la distance maximale est parmi les éléments de *voisins* alors E[i] vient prendre sa place. Sinon on passe au tour suivant de la deuxième boucle for.

2.2.1 Deuxième version avec les abscisses des points dans la liste E, k quelconque et la fonction distance définie à part.

In [ ]:
```
def proches_voisins1(E, P, k, d):
    """
    Construit la liste des k plus proches voisins de P qui sont dans
la liste E.

    Parametres nommes
    -----------------
    E : de type list
        La liste des abscisses des points des données.

    P : de type float
        P = l'abscisse du point P de l'individu exterieur a E dont on
cherche les plus proches voisins.
```

```
    k : Le nombre de plus proches voisins qu'on souhaite trouver dans
E.

    d : Fonction distance

    Retourne
    --------
    voisins : de type list
        La liste des abscisses des k plus proches voisins de P.

    """

    voisins = []
    for i in range(k):   # On initialise la liste 'voisins' avec les
elements E[0] a E[k-1]
        voisins.append(E[i])

    for i in range(k, len(E)): # On examine un a un l'element E[k], E
[k+1], ... jusqu'a la fin de E.
        dist = d(P, E[i]) # On calcule sa distance entre P et lui. On
la memorise dans 'dist'.
        u = i # On memorise dans 'u' le rang k, k+1, ... dans E de l'
element qu'on examine.

        for j in range(k):   # On compare 'dist' aux distances entre
P et tous les elements de 'voisins'.
            if  dist < d(voisins[j], P):  # Si 'dist' est inferieure
a une des distances, alors
                dist = d(voisins[j], P)  #  on mémorise cette plus co
urte distance et on memorise
                u = j  # dans 'u' le rang j de l'element de 'voisin'
qui produit une distance superieure.

        if u != i:  # Si on a trouve un element dans E plus proche de
P que l'element voisin[j] alors
            voisins[u] = E[i]  # u a change et a pris la valeur j. E[
i] vient remplacer voisin[j].
    return voisins


def d(xP, xQ):
    """
    Calcule la distance euclidienne entre :
    - le point P d'abscisse xP
    - le point Q d'abscisse xQ

    Parametres nommes
    -----------------
    x, y : de type float
        x = l'abscisse du premier point P
        y = l'abscisse du deuxieme point Q

    Retourne
    --------
    dist : de type float
        La distance euclidienne entre P et Q.

    """
```

```
    dist = abs(xQ - xP) # Sur une droite, distance = valeur absolue d
e la difference des abscisses.
    return dist

k = 3  # On cherche les 3 plus proches voisins de P
E = [1000, 200, 300, 350, 800, 80, 400, 100, 250, 780]
P = 92
print("E = ", E)
voisins = proches_voisins1(E, P, k, d)
print("voisins = ", voisins)
```

- Exécutez le code ci-dessus pour une valeur P = 92.

- Pour voir comment fonctionne cet algorithme, allez sur http://pythontutor.com/visualize.html#mode=edit (http://pythontutor.com/visualize.html#mode=edit)

Recopiez le code ci-dessus et dans la fenêtre de Python Tutor et cliquez sur le bouton **Visualize Execution**. Exécutez-le code pas à pas.

1) Combien d'étapes de calcul Python Tutor annonce-t-il lorsque E contient n = 10 éléments ?

Réponse :

- La liste E contient maintenant 20 éléments :

In [ ]:
```
def proches_voisins1(E, P, k, d):
    """
    Construit la liste des k plus proches voisins de P qui sont dans
la liste E.

    Parametres nommes
    -----------------
    E : de type list
        La liste des abscisses des points des données.

    P : de type float
        P = l'abscisse du point P de l'individu exterieur a E dont on
cherche les plus proches voisins.

    k : Le nombre de plus proches voisins qu'on souhaite trouver dans
E.
```

```
    d : Fonction distance

    Retourne
    --------
    voisins : de type list
        La liste des abscisses des k plus proches voisins de P.

    """

    voisins = []
    for i in range(k):   # On initialise la liste 'voisins' avec les elements E[0] a E[k-1]
        voisins.append(E[i])

    for i in range(k, len(E)): # On examine un a un l'element E[k], E[k+1], ... jusqu'a la fin de E.
        dist = d(P, E[i]) # On calcule sa distance entre P et lui. On la memorise dans 'dist'.
        u = i # On memorise dans 'u' le rang k, k+1, ... dans E de l'element qu'on examine.

        for j in range(k):   # On compare 'dist' aux distances entre P et tous les elements de 'voisins'.
            if  dist < d(voisins[j], P):  # Si 'dist' est inferieure a une des distances, alors
                dist = d(voisins[j], P)  #  on mémorise cette plus courte distance et on memorise
                u = j  # dans 'u' le rang j de l'element de 'voisin' qui produit une distance superieure.

        if u != i:  # Si on a trouve un element dans E plus proche de P que l'element voisin[j] alors
            voisins[u] = E[i]  # u a change et a pris la valeur j. E[i] vient remplacer voisin[j].
    return voisins


def d(xP, xQ):
    """
    Calcule la distance euclidienne entre :
    - le point P d'abscisse xP
    - le point Q d'abscisse xQ

    Parametres nommes
    -----------------
    x, y : de type float
        x = l'abscisse du premier point P
        y = l'abscisse du deuxieme point Q

    Retourne
    --------
    dist : de type float
```

```
        La distance euclidienne entre P et Q.

    """
    dist = abs(xQ - xP) # Sur une droite, distance = valeur absolue d
e la difference des abscisses.
    return dist

k = 3  # On cherche les 3 plus proches voisins de P
E = [1000, 200, 300, 350, 800, 80, 400, 100, 250, 780, 1020, 95, 331,
360, 800, 98, 400, 137, 239, 1780]
P = 92
print("E = ", E)
voisins = proches_voisins1(E, P, k, d)
print("voisins = ", voisins)
```

- Exécutez le code ci-dessus pour une valeur P = 92.

2) Combien d'étapes de calcul Python Tutor annonce-t-il lorsque E contient n = 20 éléments ?

Réponse :

- La liste E contient maintenant 30 éléments :

In [ ]:
```
def proches_voisins1(E, P, k, d):
    """
    Construit la liste des k plus proches voisins de P qui sont dans
la liste E.

    Parametres nommes
    -----------------
    E : de type list
        La liste des abscisses des points des données.

    P : de type float
        P = l'abscisse du point P de l'individu exterieur a E dont on
cherche les plus proches voisins.

    k : Le nombre de plus proches voisins qu'on souhaite trouver dans
E.

    d : Fonction distance

    Retourne
```

```
    Retourne
    --------
    voisins : de type list
        La liste des abscisses des k plus proches voisins de P.

    """

    voisins = []
    for i in range(k):   # On initialise la liste 'voisins' avec les elements E[0] a E[k-1]
        voisins.append(E[i])

    for i in range(k, len(E)): # On examine un a un l'element E[k], E[k+1], ... jusqu'a la fin de E.
        dist = d(P, E[i]) # On calcule sa distance entre P et lui. On la memorise dans 'dist'.
        u = i # On memorise dans 'u' le rang k, k+1, ... dans E de l'element qu'on examine.

        for j in range(k):   # On compare 'dist' aux distances entre P et tous les elements de 'voisins'.
            if  dist < d(voisins[j], P):  # Si 'dist' est inferieure a une des distances, alors
                dist = d(voisins[j], P)  #  on mémorise cette plus courte distance et on memorise
                u = j  # dans 'u' le rang j de l'element de 'voisin' qui produit une distance superieure.

        if u != i:  # Si on a trouve un element dans E plus proche de P que l'element voisin[j] alors
            voisins[u] = E[i]  # u a change et a pris la valeur j. E[i] vient remplacer voisin[j].
    return voisins


def d(xP, xQ):
    """
    Calcule la distance euclidienne entre :
    - le point P d'abscisse xP
    - le point Q d'abscisse xQ

    Parametres nommes
    -----------------
    x, y : de type float
        x = l'abscisse du premier point P
        y = l'abscisse du deuxieme point Q

    Retourne
    --------
    dist : de type float
        La distance euclidienne entre P et Q.
```

```
    """
    dist = abs(xQ - xP) # Sur une droite, distance = valeur absolue d
e la difference des abscisses.
    return dist

k = 3  # On cherche les 3 plus proches voisins de P
E = [1000, 200, 300, 350, 800, 80, 400, 100, 250, 780, 1020, 95, 331,
360, 800, 98, 400, 137, 239, 1780,\
    611, 322, 823, 424, 525, 1026, 91, 327, 28, 1229, 3030]
P = 92
print("E = ", E)
voisins = proches_voisins1(E, P, k, d)
print("voisins = ", voisins)
```

- Exécutez le code ci-dessus pour une valeur P = 92.

3) Combien d'étapes de calcul Python Tutor annonce-t-il lorsque E contient n = 30 éléments ?

Réponse :

4) Ces résultats sont-ils compatibles avec le fait que le coût en temps de cet algorithme est en *O(n)*?

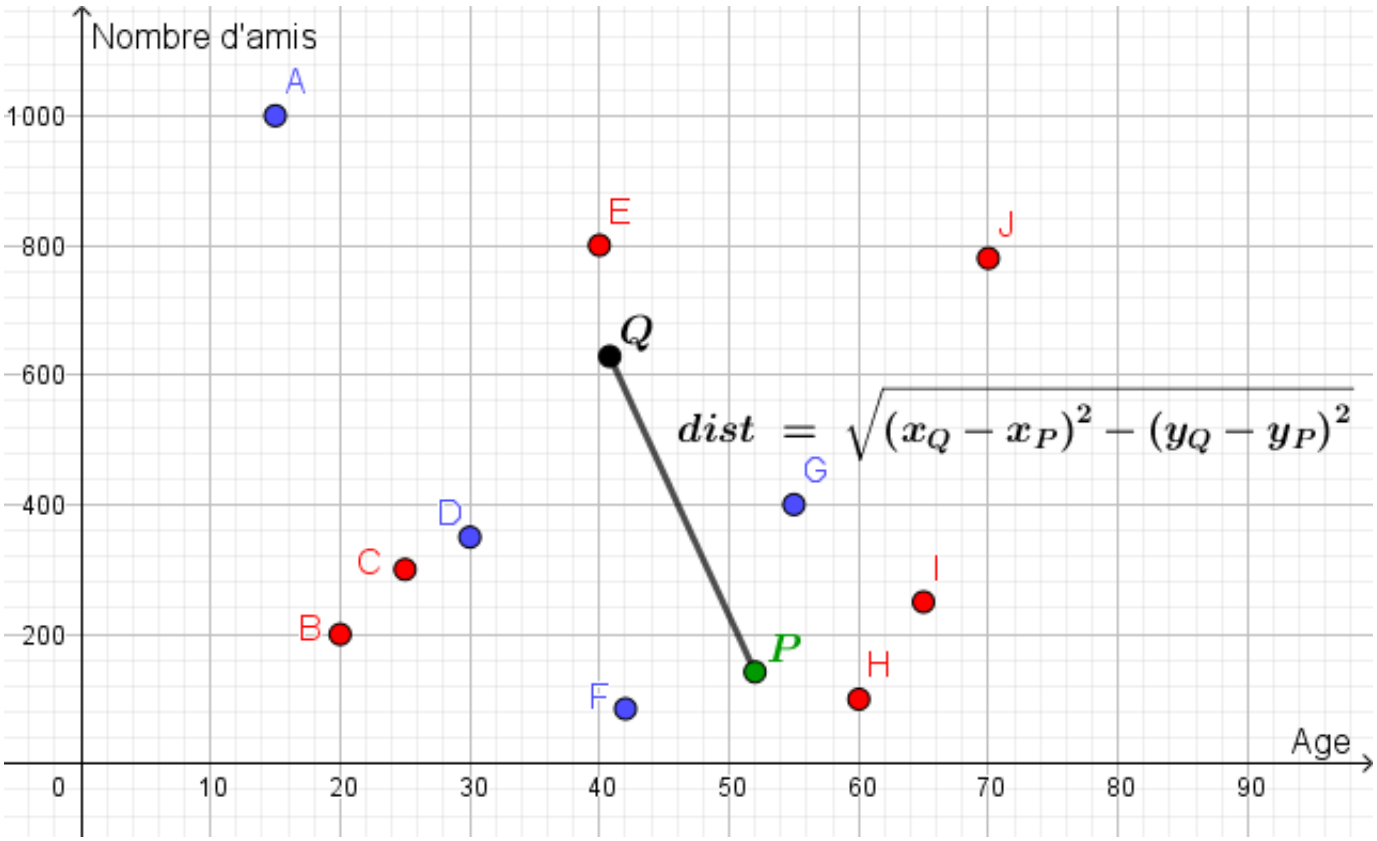
Réponse :

Lisez le début du paragraphe **Algorithme des plus proches voisins** p. 316 jusqu'au bas de la p. 317

3. Recherche des k plus proches voisins sur un tableau de dimension 2

Illustration de l'exemple de recherche des k plus proches voisins de *P* en "dimension 2" c'est à dire avec 2 coordonnees.

On a remplacé les étiquettes par un code couleur : Rouge pour "n'aime pas le film", Bleu pour "aime le film".
Vert est la couleur de l'individu extérieur aux données, dont on cherche à prédire s'il aime ou n'aime pas le film.



- A = (15, 1000)
- B = (20, 200)
- C = (25, 300)
- D = (30, 350)
- E = (40, 800)
- F = (42, 85)
- G = (55, 400)
- H = (60, 100)
- I = (65, 250)
- J = (70, 780)
- P = (52, 142)

- On a 10 points qui ont chacun **deux** coordonnées (qui symbolisent les "caractéristiques"). On connait, de plus pour chaque point, son **"étiquette"** dont la valeur pour notre exemple est bleu ou rouge.
- On a un point P dont on connait seulement les **deux** coordonnées. Pour le moment, on ne connait pas son étiquette. Dans notre exemple, son étiquette provisoire est verte.

- On peut faire la représentation graphique de points en deux dimensions avec la bibliothèque **mathplotlib** :

L'ensemble des données est : E = [(15, 1000, 'blue'), (20, 200, 'red'), (25, 300, 'red'), (30, 350, 'blue'), (40, 800, 'red'), (42, 85, 'blue'), (55, 400, 'red'), (60, 100, 'blue'), (65, 250, 'blue'), (70,

780, 'red')]

Exemple : exécutez le code ci-dessous après l'avoir examiné.

In [ ]:

```
# On définit E une liste de 11 tuples. Les tuples ont 3 elements absc
isse, ordonnee, etiquette.
# Les 10 premiers points sont des donnees dont l'etiquette est connue
'blue' ou 'red'.
# Le dernier point P a une etiquette inconnue qui est provisoirement
'green'.
# Apres l'execution de l'algorithme des k plus proches voisins, on po
urra choisir laquelle des etiquettes
# 'blue' ou 'red' on associera a P.

E = [(15, 1000, 'blue'), (20, 200, 'red'), (25, 300, 'red'), (30, 350
, 'blue'), (40, 800, 'red'), (42, 85, 'blue'),
     (55, 400, 'red'), (60, 100, 'blue'), (65, 250, 'blue'), (70, 780
, 'red'), (52, 142, 'green')]

# On appelle la bibliotheque matplotlib.
# Puis on importe le module pyplot de matplotlib pour representater g
raphiquement les 11 points de l'ensemble E.
%matplotlib inline
import matplotlib.pyplot as plt

#  Pour afficher les points avec matplotlib.pyplot, il faut une liste
pour chaque coordonnee.
liste1 = [] #  Initialisation de la liste avec la premiere coordonnee
.
liste2 = [] #  Initialisation de la liste avec la deuxieme coordonnee
.

liste3 = []  #  Initialisation de la liste avec l'etiquette.

# Extraction des coordonnees destuples
for i in range(len(E)) :
    liste1.append(E[i][0])  #  liste1 contient les premieres coordonn
ees
    liste2.append(E[i][1])  #  liste2 contient les 2emes coordonnees
    liste3.append(E[i][2])  #  liste3 contient les etiquettes

# On affiche l'espace des caractéristiques (autrement dit des coordon
nees des points)
plt.scatter(liste1, liste2, marker = 'o', color = liste3) # Affiche l
es points et leur couleur

# Affiche ne nom des axes
plt.xlabel('Age')
plt.ylabel("Nombre d'amis")

# Affiche les trois listes pour verification.
```

```
print("liste1 = ", liste1)
print("liste2 = ", liste2)
print("liste3 = ", liste3)
```

Utilisation de l'algorithme des k plus proches voisins pour determiner quelle devrait etre l'etiquette (couleur) d'un nouvel élément P.

- Regardez le code ci-dessous, en particulier les commentaires.

Exécutez-le pour un point $P(52, 142, 'green')$.

In [ ]:

```
import math  #  On utilisera math.sqrt pour le calul de la distance.

def proches_voisins2(E, P, k, d):
    """
    Construit la liste des k plus proches voisins de x qui sont dans la liste E.

    Parametres nommes
    -----------------
    E : de type list
        La liste des tuples (x_age, x_amis, couleur)

    P : de type tuple
        P = (xP, yP) le point dont on cherche les k plus proches voisins.

    k : Le nombre de plus proches voisins qu'on souhaite trouver.

    d : Fonction distance

    Retourne
    --------
    voisins : de type list de tuples
        La liste des k points plus proches voisins de x

    """

    voisins = []
    for i in range(k):  # On fait ici fonctionner l'algorithme avec k = 1.
        voisins.append(E[i]) # Au depart, la liste voisins contient le premier tuple de la liste E.

    for i in range(k, len(E)): # On parcourt tout le reste de la liste E.
        dist = d(P, E[i]) # On memorise la distance entre X et l'element E[i].
```

```
        u = i # On memorise le rang i dans E de son element qui est a
la distance dist de P.
        for j in range(k):
            if j in range (k):  # On parcourt toute la liste voisins.
                if  dist < d(voisins[j], P):  # Si la distance entre
P et E[i] est inferieure celle entre P et
                                              # l'element de rang j d
ans la liste voisins,
                    dist = d(voisins[j], P)  # alors on mémorise la v
aleur de cette plus courte distance.
                    u = j  # On memorise le rang j dans voisins de so
n element qui est a la distance dist de P
        if u != i:  # Si on a trouve un element dans E plus proche de
P que l'element voisins[j] ne l'etait
            voisins[u] = E[i]  # alors E[i] vient remplacer cet eleme
nt dans voisins.
    return voisins


def d(x, y):
    """
    Calcule la distance euclidienne entre :
    - le point P dont les coordonnees sont les deux premiers elements
du tuple x
    - le point Q dont les coordonnees sont les deux premiers elements
du tuple y.

    Parametres nommes
    -----------------
    x, y : de type tuple
        x(x[0], x[1], x[2]) donne les coordonnees du premier point P(
x[0], x[1])
        y(y[0], y[1], y[2]) donne les coordonnees du deuxieme point Q
(y[0], y[1])

    Retourne
    --------
    dist : de type float
        La distance euclidienne entre P et Q.

    """
    xP = x[0]
    yP = x[1]
    xQ = y[0]
    yQ = y[1]

    dist = math.sqrt((xQ - xP)**2 + (yQ - yP)**2)
    return dist

k = 3
E = [[15, 1000, 'blue'], [20, 200, 'red'], [25, 300, 'red'], [30, 350
  'blue'], [40, 800, 'red'], [42, 85, 'blue'],
```

```
     [55, 400, 'red'], [60, 100, 'blue'], [65, 250, 'blue'], [70, 780
, 'red']]   #   E est la liste des features.
P = (52, 142, 'green')
print("E = ", E)
print()
print("Dans E, les plus proches voisins de P = ", P, "sont : ", proch
es_voisins2(E, P, k, d))
```

5) Quel sont les 3 points les plus proches de P ? (Donnez leurs n-uplets)

Réponse :

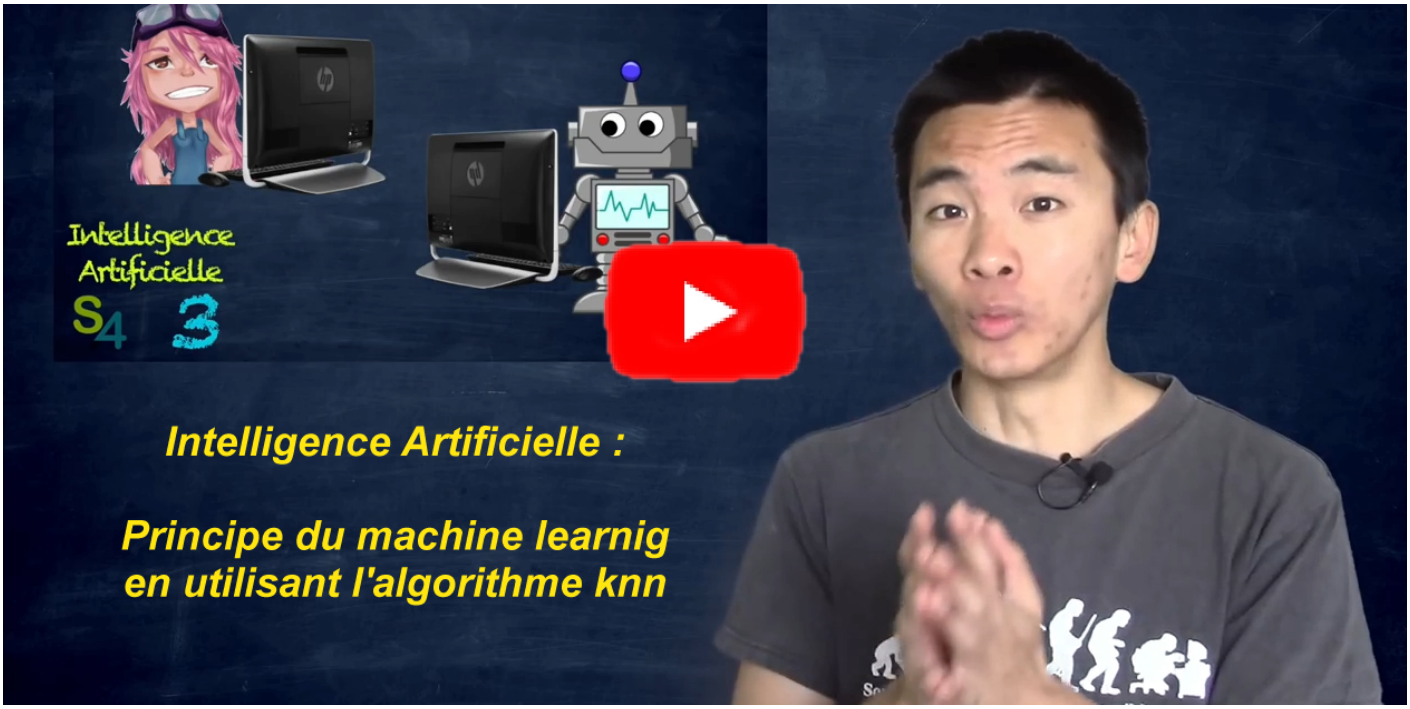
6) Quelle est la couleur de l'étiquette qu'on peut attribuer à P selon l'algorithme des k plus proches voisins lorsque k = 3 ? Justifiez votre réponse.

Réponse :

En intelligence artificielle, Les algorithmes d'apprentissage peuvent utiliser plusieurs modes d'apprentissage. On s'intéresse ici à l'apprentissage supervisé.

- Si les classes sont prédéterminées et les exemples connus, le système apprend à classer selon un modèle de classement ; on parle alors d'apprentissage supervisé (ou d'analyse discriminante).
- Un expert doit préalablement étiqueter des exemples. Le processus se passe en deux phases.
- Lors de la première phase (hors ligne, dite d'apprentissage), il s'agit de déterminer un modèle à partir des données étiquetées.
- La seconde phase (en ligne, dite de test) consiste à prédire l'étiquette d'une nouvelle donnée, connaissant le modèle préalablement appris.

Regardez la vidéo ci-dessous extraite de la chaine YouTube de Science4All (https://www.youtube.com/channel/UC0NCbj8CxzeCGIF6sODJ-7A) :



(http://www.astrovirtuel.fr/jupyter/19_pnsi_cours/voisins.mp4)

4. Exemple : La machine apprend les catégories de fruits. Puis elle est capable d'en reconnaitre un nouveau.

Algorithme des *k* plus proches voisins en triant d'abord les données selon les distances

L'algorithme k-nn se base sur un jeu de données comportant plusieurs paramètres les *features* (qui se mesurent à l'aide d'un nombre), et un paramètre qualitatif : la classe de l'objet ou *label*.

L'objectif de l'algorithme est de classer un nouvel individu (au sens statistique) dans une des classes de la population existante.

Déroulement de l'algorithme :

- Calcul de toutes les distances euclidiennes entre le jeu de données connues et le nouvel individu.
- Tri des éléments du jeu de données par les distances dans l'ordre croissant
- Sélection des k voisins les plus proches : ce sont les k premiers éléments triés à l'étape précédente
- Le nouvel individu appartient à la classe majoritaire dans les k plus proches voisins

Mise en forme des données

Nous allons travailler sur de la reconnaissance de fruits.

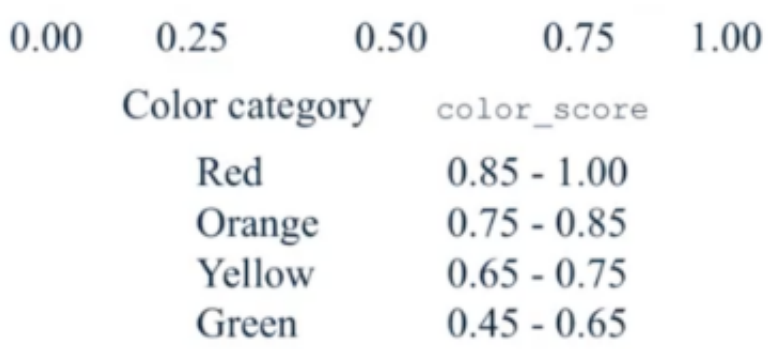
Les données se trouvent dans le fichier donnees_fruits.csv (http://www.astrovirtuel.fr/jupyter/19_pnsi_cours/donnees_fruits.csv), à enregistrer dans le dossier où ce trouve ce jupyter notebook.

Ce fichier csv contient la table suivante :

	A	B	C	D	E
1	nom_fruit	masse	largeur	hauteur	score_couleur
2	pomme	192	8.4	7.3	0.55
3	pomme	180	8	6.8	0.59
4	pomme	176	7.4	7.2	0.6
5	mandarine	86	6.2	4.7	0.8
6	mandarine	84	6	4.6	0.79
7	mandarine	80	5.8	4.3	0.77
8	mandarine	80	5.9	4.3	0.81
9	mandarine	76	5.8	4	0.81
10	pomme	178	7.1	7.8	0.92
11	pomme	172	7.4	7	0.89
12	pomme	166	6.9	7.3	0.93
13	pomme	172	7.1	7.6	0.92
14	pomme	154	7	7.1	0.88
15	pomme	164	7.3	7.7	0.7
16	pomme	152	7.6	7.3	0.69
17	pomme	156	7.7	7.1	0.69
18	pomme	156	7.6	7.5	0.67
19	pomme	168	7.5	7.6	0.73
20	pomme	162	7.5	7.1	0.83
21	pomme	162	7.4	7.2	0.85
22	pomme	160	7.5	7.5	0.86
23	pomme	156	7.4	7.4	0.84
24	pomme	140	7.3	7.1	0.87
25	pomme	170	7.6	7.9	0.88
26	orange	342	9	9.4	0.75
27	orange	356	9.2	9.2	0.75
28	orange	362	9.6	9.2	0.74
29	orange	204	7.5	9.2	0.77
30	orange	140	6.7	7.1	0.72

31	orange	160	7	7.4	0.81
32	orange	158	7.1	7.5	0.79
33	orange	210	7.8	8	0.82
34	orange	164	7.2	7	0.8
35	orange	190	7.5	8.1	0.74
36	orange	142	7.6	7.8	0.75
37	orange	150	7.1	7.9	0.75
38	orange	160	7.1	7.6	0.76
39	orange	154	7.3	7.3	0.79
40	orange	158	7.2	7.8	0.77
41	orange	144	6.8	7.4	0.75
42	orange	154	7.1	7.5	0.78
43	orange	180	7.6	8.2	0.79
44	orange	154	7.2	7.2	0.82
45	citron	194	7.2	10.3	0.7
46	citron	200	7.3	10.5	0.72
47	citron	186	7.2	9.2	0.72
48	citron	216	7.3	10.2	0.71
49	citron	196	7.3	9.7	0.72
50	citron	174	7.3	10.1	0.72
51	citron	132	5.8	8.7	0.73
52	citron	130	6	8.2	0.71
53	citron	116	6	7.5	0.72
54	citron	118	5.9	8	0.72
55	citron	120	6	8.4	0.74
56	citron	116	6.1	8.5	0.71
57	citron	116	6.3	7.7	0.72
58	citron	116	5.9	8.1	0.73
59	citron	152	6.5	8.5	0.72
60	citron	118	6.1	8.1	0.7
61					

The scale for the (simplistic) `color_score` feature used in the fruit dataset



Le "score_couleur" est un flottant sur [0 ; 1] qui mesure la couleur. Vers 0 les couleurs "froides" d'un coté du spectre des couleurs, et vers 1 les couleurs "chaudes" de l'autre côté du spectre.

- Le programme suivant importe un fichier csv, et renvoie une *liste de dictionnaires*.
- Chaque dictionnaire correspond à une ligne de la table.
- Chaque dictionnaire a comme items des couples

```
noms des champs : valeur
```

In [ ]:
```
import csv
from collections import OrderedDict   # structure de données utilisée

# On importe le fichier d'exemple de base de données de fruits
donnees_fruits_csv = open('donnees_fruits.csv','r', encoding ='utf-8'
)
lecteur = csv.DictReader(donnees_fruits_csv, delimiter=';')
fruits = list(lecteur)

print("Identifiants des champs de données et type du champ")
items_fruits =list(fruits[0].items())
data = list(fruits[0].values())
for champ,donnee in items_fruits:
    print("L'identifiant '",champ,"' comporte des données de type ",t
ype(donnee),", de valeur pour le 1er fruit :",
          donnee)

donnees_fruits_csv.close()
```

On retrouve le problème déjà constaté dans la séquence 11 sur les données : les champs

numériques sont considérés comme du texte.
Dans la cellule ci-dessous, on s'inspire de l'algorithme de la séquence 11, question 51 et on modifie le programme précédent pour que les champs aient le bon type de données.

- On voit qu'il est aussi nécessaire, avant de convertir une variable de type string en variable de type int ou float de lui enlever les caractères "guillemets" qui l'entourent.
- Cela est réalisé par la méthode

```
variable.strip('"')
```

Note :

Pour enlever les guillemets avant ma_chaine :

```
ma_chaine.lstrip('"')
```

Pour enlever les guillemets apr-st ma_chaine :

```
ma_chaine.rstrip('"')
```

Pour enlever les guillemets avant et après ma_chaine :

```
ma_chaine.strip('"')
```

In [ ]:
```
ma_variable = '"1.143"'
ma_variable_sans_guillemets = ma_variable.strip('"')

print(ma_variable)
print(ma_variable_sans_guillemets)
print(type(ma_variable_sans_guillemets))
```

In [ ]:
```
mon_flottant = float(ma_variable_sans_guillemets)
print(mon_flottant)
print(type(mon_flottant))
```

In [ ]:
```
#  Pour importer une table avec Modification de type pour les champs
contenant des nombres entiers
f = open("donnees_fruits.csv", "r", encoding = "utf-8")
champs = f.readline() #  Lecture de la première ligne
lignes = f.readlines() #  Lecture des autres lignes
table=[]

champs2 = champs.rstrip().split(';') # Met les champs dans une liste
print("champs2 avant = ", champs2)

champs2[0] = str(champs2[0]).strip('"')  #  .strip('"') retire le car
actère " si trouvé à gauche ou à droite.
champs2[1] = str(champs2[1]).strip('"')
```

```
champs2[2] = str(champs2[2]).strip('"')
champs2[3] = str(champs2[3]).strip('"')
champs2[4] = str(champs2[4]).strip('"')


for ligne in lignes:
    liste = ligne.rstrip().split(';')
    liste[0] = str(liste[0]).strip('"')
    liste[1] = int(liste[1])  # Modification de domaine de valeur pou
r transformer une chaine de caractères en entier
    liste[2] = float(str(liste[2]).strip('"'))
    liste[3] = float(str(liste[3]).strip('"'))
    liste[4] = float(str(liste[4]).strip('"'))
    table.append(liste)
f.close()

print("champs2 après = ", champs2)
print()
print("table = ", table)
```

7) Dans la cellule ci-dessous, écrivez un programme qui :
A partir de :

champs2 = ['nom_fruit', 'masse', 'largeur', 'hauteur', 'score_couleur']

et de :

table = [['pomme', 192, 8.4, 7.3, 0.55], ['pomme', 180, ...

renvoie la liste de dictionnaires :

fruits = [{'nom_fruit': 'pomme', 'masse': 192.0, 'largeur': 8.4, 'hauteur': 7.3, 'score_couleur': 0.55}, {'nom_fruit': 'pomme', 'masse': 180.0, ...

In [ ]:
```
from copy import deepcopy

ma_copie = deepcopy(table)

fruits = []  #  fruits sera la même liste de dictionnaires que fruit_
texte, mais la masse, la largeur, la hauteur
             #  et l'index_couleur seront des flottants
```

```
#  pour chaque ligne dans table
for ligne in table:

    #  Crée un dictionnaire
    dico = {}

    #  remplit le dictionnaire
    ...
    ...
    ...
    ...
    ...
    fruits.append(dico)

print("fruits = ", fruits, "\n")  #  fruits_texte est une liste de di
ctionnaires dont chaque
                                  #  dictionnaire représente comme un
"p-uplet nommé" (L'objet namedtuple
                                  #  existe dans collections dans Pyt
hon mais est hors programme).
```

8) Pour la suite, il est nécessaire de savoir quels sont les types de fruits (ananas, kiwis, etc.) présents dans la base de données. Ecrire un programme qui répond à cette question.

In [ ]:
```
sortes_de_fruits_au_total = []  #  Contiendra les sortes de fruits di
fférentes présentes dans la table.

for ligne in fruits:
    ...
    ...
    ...

print("sortes_de_fruits_au_total = ", sortes_de_fruits_au_total)
```

Présentation des données

- Le programme suivant importe la bibliothèque matplotlib, qui permet de faire assez facilement tout type de graphique scientifique.
- Il donne ensuite tous les schémas possibles en deux dimensions, avec les 4 champs numériques.
- Les pommes sont en vert, les citrons en bleu (pour des raisons de lisibilité), les oranges en orange et les mandarines en rouge. *Eventuellement cliquez sur les schémas, pour les*

voir tous

In [ ]:

```
import matplotlib.pyplot as plt
# import matplotlib.patches as mpatches
from matplotlib.lines import Line2D

#Données
nom = [fruit['nom_fruit'] for fruit in fruits]
code_couleur = []
code_marqueur = []
for i in range(len(nom)):
    if nom[i] == "pomme":
        code_couleur.append('green')
        code_marqueur.append('.')
    elif nom[i] == 'mandarine':
        code_couleur.append('red')
        code_marqueur.append('v')
    elif nom[i] == 'orange':
        code_couleur.append('orange')
        code_marqueur.append('s')
    else:
        code_couleur.append('blue')
        code_marqueur.append('*')
masses =[fruit['masse'] for fruit in fruits]
largeurs = [fruit['largeur'] for fruit in fruits]
hauteurs = [fruit['hauteur'] for fruit in fruits]
couleurs = [fruit['score_couleur'] for fruit in fruits]

#graphiques
#subplot permet de faire plusieurs graphiques (ici 3 lignes 2 colonne
s, on précise la taille du schéma total)
fig, axes= plt.subplots(nrows=3, ncols=2,figsize=(18,18))

#pour la légende, compliquée à faire
legend_elements = [Line2D([0], [0], marker='o', color='w', label='Pom
mes', markerfacecolor='g', markersize=10),
                   Line2D([0], [0], marker='o', color='w', label='Man
darines', markerfacecolor='r', markersize=10),
                   Line2D([0], [0], marker='o', color='w', label='Ora
nges', markerfacecolor='orange', markersize=10),
                   Line2D([0], [0], marker='o', color='w', label='Cit
rons', markerfacecolor='blue', markersize=10),
                  ]

plt.subplot(321)    # sous-schéma 1 dans les 3 colonnes et 2 lignes (
3 - 2 - 1)
plt.scatter(masses, largeurs, c = code_couleur, alpha = 0.5, s = 70)
plt.xlabel('masse')
plt.ylabel('largeur')
plt.legend(handles=legend_elements,loc = 'lower right')
```

```
"""
# Autre méthode pour la légende, avec des rectangles
classes = ['pomme', 'mandarine', 'orange', 'citron']
class_colours = ['green','red','orange','yellow']
ronds = []
for i in range(0,len(class_colours)):
    ronds.append(mpatches.Circle((0,0),5, color = class_colours[i]))
# Circle fait des rectangles O_o
plt.legend(ronds,classes)
"""

plt.subplot(322)
plt.scatter(masses, hauteurs, c = code_couleur, alpha = 0.5, s = 70)
plt.xlabel('masse')
plt.ylabel('hauteur')
plt.legend(handles=legend_elements,loc = 'lower right')

plt.subplot(323)
plt.scatter(masses, couleurs, c = code_couleur, alpha = 0.5, s = 70)
plt.xlabel('masse')
plt.ylabel('score couleur')
plt.legend(handles=legend_elements,loc = 'lower right')

plt.subplot(324)
plt.scatter(largeurs, hauteurs, c = code_couleur, alpha = 0.5, s = 70
)
plt.xlabel('largeur')
plt.ylabel('hauteur')
plt.legend(handles=legend_elements,loc = 'lower right')

plt.subplot(325)
plt.scatter(largeurs, couleurs, c = code_couleur, alpha = 0.5, s = 70
)
plt.xlabel('largeur')
plt.ylabel('score couleur')
plt.legend(handles=legend_elements,loc = 'lower left')

plt.subplot(326)
plt.scatter(hauteurs, couleurs, c = code_couleur, alpha = 0.5, s = 70
)
plt.xlabel('hauteur')
plt.ylabel('score couleur')
plt.legend(handles=legend_elements,loc = 'lower left')


plt.show()
```

Conclusion graphique:

Les schémas précédents montrent, suivant les projections, une classification nette pour les

mandarines, assez nette pour les citrons, mais bien moins claire pour les pommes et oranges.

Classification de nouveaux individus

On donne deux individus de classe inconnue, dont on a mesuré les caractéristiques suivantes (masse, largeur, hauteur) :

```
fruit1 = (100, 6.3, 8.0)

fruit2 = (162, 7.1, 7.3)
```

On voudrait savoir à quelle classe appartiennent ces fruits. Pour cela on va programmer l'algorithme *k*-nn, suivant la méthode exposée en préambule :

- Calcul de toutes les distances euclidiennes entre le jeu de données connues et le nouvel individu.
- Tri des éléments du jeu de données par les distances dans l'ordre croissant.
- Sélection des k voisins les plus proches : ce sont les k premiers éléments triés à l'étape précédente. On pourra prendre k = 3 ou k = 5 comme proposé ci-dessous.
- Le nouvel individu appartient à la classe majoritaire dans les k plus proches voisins.

9) Compléter le programme suivant :

In [ ]:
```
# Programme k-nn

from math import sqrt
from copy import deepcopy

def distance(pointA, pointB):
    """
    Renvoie la distance euclidienne entre deux points. Un point est v
u comme un tuple.


    Parametres nommes :
    -------------------
    pointA :   tuple de flottants. Ex : (192, 8.4, 7.3)

    pointB :   tuple de flottants, de même dimension que pointA. Ex :
(180, 8.0, 6.8)


    Retourne :
    ----------
```

```
    distAB :  flottant positif ou nul, distance euclidienne entre A et B. Ex : 12.017

    """




    return distAB



def kPlusProches(point, liste, k):
    """
    Renvoie la liste des k plus proches voisins d'un point dans une liste de données.

    Parametres nommes :
    -------------------
    point : tuple de flottants.  Ex : (100, 6.3, 8.0)

    liste : liste de tuples, les tuples ont comme dimension celle de point + 1. La première
            valeur de chaque tuple est la classe de l'individu, les autres sont les
            coordonnées (valeurs des paramètres). Ex :
            [('pomme', 192.0, 8.4, 7.3), ('pomme', 180.0, 8.0, 6.8), ...]

    k :     entier strictement positif, inférieur ou égal au nombre d'éléments de liste.

    Retourne :
    ----------
    k_voisins : liste de tuples, les tuples ont comme dimension celle de point + 1.
                liste des voisins les plus proches de "point" dans "liste", au sens de
                la distance appelée dans cette fonction. Ex : [('pomme', 192.0, 8.4, 7.3),
                ('mandarine', 76.0, 5.8, 4.0), ('pomme', 178.0, 7.1, 7.8)]

    """




    return k_voisins
```

```
def appartient_a(point, liste, k):
    """
    Renvoie la classe d'un individu déterminée par celle de ses k plus proches voisins.

    Parametres nommes :
    -------------------
    point : tuple de flottants. Ex : (100, 6.3, 8.0)

    liste : liste de tuples, les tuples ont comme dimension celle de point + 1. La première
            valeur de chaque tuple est la classe de l'individu, les autres sont les
            coordonnées (valeurs des paramètres).

    k :     entier strictement positif, inférieur ou égal au nombre d'éléments de liste.


    Retourne :
    ----------
    classe : chaine de caractères donnant la classe de "point". C'est la classe majoritaire dans les
             k-voisins les plus proches.

    """




    return classe




#  Rassemblement des 3 paramètres en vue de l'exécution de la fonction
#  appartient_a(point, liste, k)
#  qui renvoie la classe majoritaire à laquelle apparinet vraisemblablement 'point'.



fruit1 = (100, 6.3, 8.0)
fruit2 = (162, 7.1, 7.3)
mon_point = fruit2
```

```
#  Fabrication de ma_liste = [('pomme', 192.0, 8.4, 7.3), ('pomme', 1
80.0, 8.0, 6.8), ...]
#  à partir de la liste de dictionnaires
#  fruits = [{'nom_fruit': 'pomme', 'masse': 192, 'largeur': 8.4, 'ha
uteur': #  7.3, 'score_couleur': 0.55},
#            {'nom_fruit': 'pomme', 'masse': 180, 'largeur': 8.0, ...
}, ...]
ma_liste = []
for ligne in fruits:
    ma_liste.append(tuple(ligne.values()))



nombre_de_voisins = 5



ma_classe = appartient_a(mon_point, ma_liste, nombre_de_voisins)
print("La classe du fruit est probablement :", ma_classe)
```

10) Que concluez-vous ?

Réponse :

Un graphique en 3D

Ci-dessous un graphique 3D qui montre fruit1 **dans les citrons** et fruit2 **entre pommes et citrons** (dans la fenêtre du graphic User interface (GUI) qui s'ouvre, maintenez le bouton de la souris enfoncé pour faire tourner la figure 3D).

In [ ]:
```
%matplotlib tk
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

#graphiques
fig = plt.figure(figsize=(9,9))
ax = Axes3D(fig)

#pour la légende, compliquée à faire
legend_elements = [Line2D([0], [0], marker='o', color='w', label='Pom
```

```
mes', markerfacecolor='g', markersize=10),
                   Line2D([0], [0], marker='*', color='w', label='Man
darines', markerfacecolor='r', markersize=10),
                   Line2D([0], [0], marker='s', color='w', label='Ora
nges', markerfacecolor='orange', markersize=10),
                   Line2D([0], [0], marker='p', color='w', label='Cit
rons', markerfacecolor='blue', markersize=10),
                   Line2D([0], [0], marker='v', color='w', label='Inc
onnu', markerfacecolor='black', markersize=10),
                  ]

#fig.add_subplot(111, projection = '3d')
for i in range(len(nom)):
    if nom[i] == "pomme":
        ax.scatter(masses[i], largeurs[i], hauteurs[i], c="green", s=
70, alpha=0.5)
    elif nom[i] == 'mandarine':
        ax.scatter(masses[i], largeurs[i], hauteurs[i], c="red", s=70
, marker='*', alpha=0.5)
    elif nom[i] == 'orange':
        ax.scatter(masses[i], largeurs[i], hauteurs[i], c="orange", s
=70, marker='s', alpha=0.5)
    else:
        ax.scatter(masses[i], largeurs[i], hauteurs[i], c="blue", s=7
0, marker='p', alpha=0.5)
#ax.scatter(masses, largeurs, hauteurs, c = code_couleur, s= 70)
fruit1  = [100, 6.3, 8.0]
fruit2 = [162, 7.1, 7.3]
ax.scatter(100, 6.3, 8.0, c='black', marker="v", s=110)  #  fruit1 es
t le petit triangle noir
ax.scatter(162, 7.1, 7.3, c='black', marker="v", s=220)  #  fruit2 es
t le gros triangle noir
ax.set_xlabel('masse')
ax.set_ylabel('largeur')
ax.set_zlabel('hauteur')
fig.legend(handles=legend_elements,loc = 'lower right')

plt.show()
```