

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

Table des matières

[1. Introduction](#)

[2. Recherche des k plus proches voisins sur un tableau de dimension 1](#)

- [2.1 Explication de l'algorithme des k plus proches voisins avec un diaporama](#)
- [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](#)
- [2.2.2 Deuxième version avec les abscisses des points dans la liste E, k quelconque et la fonction distance définie à part](#)

[3. Recherche des k plus proches voisins sur un tableau de dimension 2](#)

[4. Exemple : La machine apprend les catégories de fruits. Puis elle est capable d'en reconnaître un nouveau.](#)

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 connaît :

- Sa caractéristique ("feature"). Ici dans l'exemple, c'est $x' = \text{nombre d'amis sur Facebook}$
- 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 connaît que la "feature" : il a 92 amis Facebook.

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).

- 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 traillons avec $k = 3$.
- Donc on fera dans l'ordre :
 - 1. La liste des distances entre P et chacun des 10 points.
 - 2. Extraire la sous-liste des 3 plus petites distances (autrement dit la liste des trois plus proches voisins de P).
 - 3. 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.
- et 2. forment la phase d'apprentissage supervisé c'est à dire l'apprentissage à partir d'exemples connus ou "expertisés".
- 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).

Principe de l'algorithme de recherche des 3 plus proches voisins

Ou ce qui revient au même, recherche des 3 plus petites distances parmi les éléments de E.

$j = 0 \quad 1 \quad 2$

Liste « voisins »
(les 3 plus petites distances)

$i = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5$

Liste E
La liste des points. Pour simplifier, on a ici les distances entre P et les points.

Menu 00:01:00:00:23 01/23

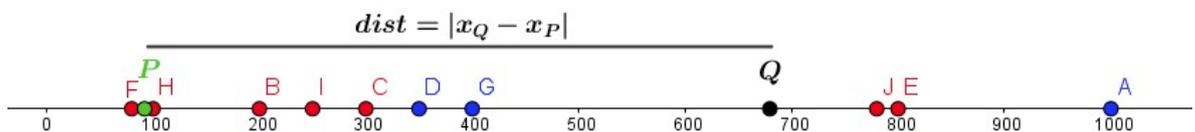
http://www.astrovirtuel.fr/jupyter/19_pnsi_cours/principe_k_plus_proches_voisins/principe_k_plus_proches_voisins.html

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)

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				

```

• 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

```

Algorithme des k plus proches voisins

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

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.

```

Entrée [ ]: 1 # Dans un premier temps, pour simplifier, on suppose qu'on a déjà calculé le
2 # entre P et les 10 points A, B, ..., J.
3 # Donc E = [908, 108, 208, 258, 708, 12, 308, 8, 158, 688].
4
5 # On cherche les k = 3 plus proches voisins de P, c'est à dire les 3 plus pe
6 # A la fin du programme la liste voisins contiendra les 3 plus petits elemen
7
8
9
10 E = [908, 108, 208, 258, 708, 12, 308, 8, 158, 688]
11 voisins = []
12
13 # Initialisation de la liste voisins avec les 3 premiers elements de E.
14 for i in range(3):
15     voisins.append(E[i])
16

```

```

17 for i in range(3, 10):
18     dist = E[i] # Initialisation de dist =
19     u = i # On stocke dans u l'indice i correspondant a dist
20
21     # Recherche du max (E[i], voisins[0], voisins[1], voisins[2]).
22     for j in range(3):
23         if voisins[j] > dist:
24             dist = voisins[j]
25             u = j # On stocke dans u l'indice j correspondant a dist
26
27     if u != i: # Si l'indice du max correspond a un element de voisins
28         voisins[u] = E[i] # alors on remplace cet element par E[i]
29
30 print("E = ", E)

```

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 été 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.

```

Entrée [ ] :
1 def proches_voisins1(E, P, k, d):
2     """
3     Construit la liste des k plus proches voisins de P qui sont dans la list
4
5     Parametres nommes
6     -----
7     E : de type list
8         La liste des abscisses des points des données.
9
10    P : de type float
11        P = l'abscisse du point P de l'individu exterieur a E dont on cherch
12
13    k : Le nombre de plus proches voisins qu'on souhaite trouver dans E.
14
15    d : Fonction distance
16
17    Retourne
18    -----
19    voisins : de type list
20        La liste des abscisses des k plus proches voisins de P.
21
22    """
23
24    voisins = []
25    for i in range(k): # On initialise la liste 'voisins' avec les element
26        voisins.append(E[i])
27
28    for i in range(k, len(E)): # On examine un a un l'element E[k], E[k+1],
29        dist = d(P, E[i]) # On calcule sa distance entre P et lui. On la mem
30        u = i # On memorise dans 'u' le rang k, k+1, ... dans E de l'element
31
32        for j in range(k): # On compare 'dist' aux distances entre P et to
33            if dist < d(voisins[j], P): # Si 'dist' est inferieure a une d

```

```

34         dist = d(voisins[j], P) # on mémorise cette plus courte di
35         u = j # dans 'u' le rang j de l'element de 'voisin' qui pro
36
37         if u != i: # Si on a trouve un element dans E plus proche de P que
38             voisins[u] = E[i] # u a change et a pris la valeur j. E[i] vien
39     return voisins
40
41
42 def d(xP, xQ):
43     """
44     Calcule la distance euclidienne entre :
45     - le point P d'abscisse xP
46     - le point Q d'abscisse xQ
47
48     Parametres nommes
49     -----
50     x, y : de type float
51         x = l'abscisse du premier point P
52         y = l'abscisse du deuxieme point Q
53
54     Retourne
55     -----
56     dist : de type float
57         La distance euclidienne entre P et Q.
58
59     """
60     dist = abs(xQ - xP) # Sur une droite, distance = valeur absolue de la di
61     return dist
62
63 k = 3 # On cherche les 3 plus proches voisins de P
64 E = [1000, 200, 300, 350, 800, 80, 400, 100, 250, 780]
65 P = 92
66 print("E = ", E)
67 voisins = proches_voisins1(E, P, k, d)

```

- 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 :

Entrée []:

```

1 def proches_voisins1(E, P, k, d):
2     """
3     Construit la liste des k plus proches voisins de P qui sont dans la list
4
5     Parametres nommes
6     -----
7     E : de type list
8         La liste des abscisses des points des données.
9
10    P : de type float
11        P = l'abscisse du point P de l'individu exterieur a E dont on cherch
12
13    k : Le nombre de plus proches voisins qu'on souhaite trouver dans E.

```

```

14
15     d : Fonction distance
16
17     Retourne
18     -----
19     voisins : de type list
20             La liste des abscisses des k plus proches voisins de P.
21
22     """
23
24     voisins = []
25     for i in range(k): # On initialise la liste 'voisins' avec les element
26         voisins.append(E[i])
27
28     for i in range(k, len(E)): # On examine un a un l'element E[k], E[k+1],
29         dist = d(P, E[i]) # On calcule sa distance entre P et lui. On la mem
30         u = i # On memorise dans 'u' le rang k, k+1, ... dans E de l'element
31
32         for j in range(k): # On compare 'dist' aux distances entre P et to
33             if dist < d(voisins[j], P): # Si 'dist' est inferieure a une d
34                 dist = d(voisins[j], P) # on memorise cette plus courte di
35                 u = j # dans 'u' le rang j de l'element de 'voisin' qui pro
36
37         if u != i: # Si on a trouve un element dans E plus proche de P que
38             voisins[u] = E[i] # u a change et a pris la valeur j. E[i] vien
39     return voisins
40
41
42     def d(xP, xQ):
43         """
44         Calcule la distance euclidienne entre :
45         - le point P d'abscisse xP
46         - le point Q d'abscisse xQ
47
48         Parametres nommes
49         -----
50         x, y : de type float
51             x = l'abscisse du premier point P
52             y = l'abscisse du deuxieme point Q
53
54         Retourne
55         -----
56         dist : de type float
57             La distance euclidienne entre P et Q.
58
59         """
60         dist = abs(xQ - xP) # Sur une droite, distance = valeur absolue de la di
61         return dist
62
63     k = 3 # On cherche les 3 plus proches voisins de P
64     E = [1000, 200, 300, 350, 800, 80, 400, 100, 250, 780, 1020, 95, 331, 360, 8
65     P = 92
66     print("E = ", E)
67     voisins = proches_voisins1(E, P, k, d)

```

- 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 :

Entrée []:

```
1 def proches_voisins1(E, P, k, d):
2     """
3     Construit la liste des k plus proches voisins de P qui sont dans la list
4
5     Parametres nommes
6     -----
7     E : de type list
8         La liste des abscisses des points des données.
9
10    P : de type float
11        P = l'abscisse du point P de l'individu exterieur a E dont on cherch
12
13    k : Le nombre de plus proches voisins qu'on souhaite trouver dans E.
14
15    d : Fonction distance
16
17    Retourne
18    -----
19    voisins : de type list
20        La liste des abscisses des k plus proches voisins de P.
21
22    """
23
24    voisins = []
25    for i in range(k): # On initialise la liste 'voisins' avec les element
26        voisins.append(E[i])
27
28    for i in range(k, len(E)): # On examine un a un l'element E[k], E[k+1],
29        dist = d(P, E[i]) # On calcule sa distance entre P et lui. On la mem
30        u = i # On memorise dans 'u' le rang k, k+1, ... dans E de l'element
31
32        for j in range(k): # On compare 'dist' aux distances entre P et to
33            if dist < d(voisins[j], P): # Si 'dist' est inferieure a une d
34                dist = d(voisins[j], P) # on memorise cette plus courte di
35                u = j # dans 'u' le rang j de l'element de 'voisin' qui pro
36
37        if u != i: # Si on a trouve un element dans E plus proche de P que
38            voisins[u] = E[i] # u a change et a pris la valeur j. E[i] vien
39    return voisins
40
41
42 def d(xP, xQ):
43     """
44     Calcule la distance euclidienne entre :
45     - le point P d'abscisse xP
46     - le point Q d'abscisse xQ
47
48     Parametres nommes
49     -----
50     x, y : de type float
51         x = l'abscisse du premier point P
52         y = l'abscisse du deuxieme point Q
53
54     Retourne
55     -----
56     dist : de type float
57         La distance euclidienne entre P et Q.
58
59     """
60     dist = abs(xQ - xP) # Sur une droite, distance = valeur absolue de la di
61     return dist
62
63 k = 3 # On cherche les 3 plus proches voisins de P
64 E = [1000, 200, 300, 350, 800, 80, 400, 100, 250, 780, 1020, 95, 331, 360, 8
65     611, 322, 823, 424, 525, 1026, 91, 327, 28, 1229, 3030]
66 P = 92
67 print("E = ", E)
68 voisins = proches_voisins1(E, P, k, d)
```

- 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 :

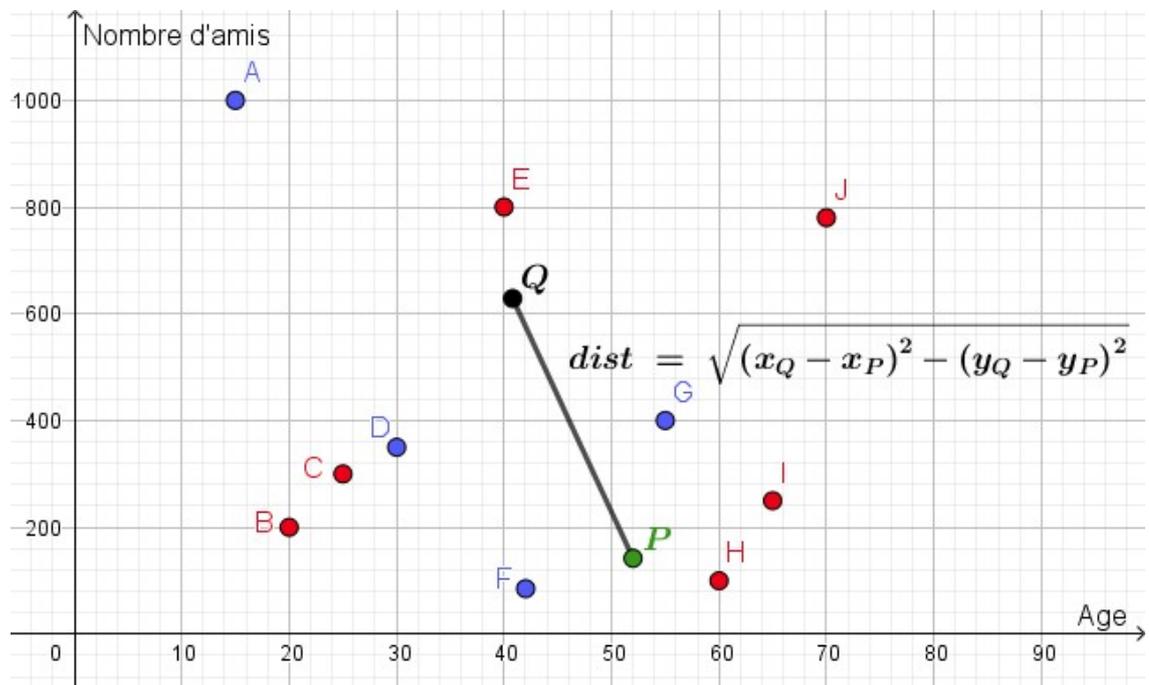
Lisez le début du paragraphe **Algorithme des plus proches voisins** p. 316 jusqu'au bas de la p. 317

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 coordonnées.

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)**

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

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.

```
Entrée [ ] : 1 # On définit E une liste de 11 tuples. Les tuples ont 3 elements abscisse, ordonnee, et
2 # Les 10 premiers points sont des donnees dont l'etiquette est connue 'blue'
3 # Le dernier point P a une etiquette inconnue qui est provisoirement 'green'
4 # Apres l'execution de l'algorithme des k plus proches voisins, on pourra choisir
5 # 'blue' ou 'red' on associera a P.
6
7 E = [(15, 1000, 'blue'), (20, 200, 'red'), (25, 300, 'red'), (30, 350, 'blue'),
8       (40, 800, 'red'), (42, 85, 'blue'), (55, 400, 'red'), (60, 100, 'blue'), (65, 250, 'blue'), (70, 780, 'red')]
9
10 # On appelle la bibliotheque matplotlib.
11 # Puis on importe le module pyplot de matplotlib pour representater graphique
12 %matplotlib inline
13 import matplotlib.pyplot as plt
14
15 # Pour afficher les points avec matplotlib.pyplot, il faut une liste pour chaque coordonnee.
16 liste1 = [] # Initialisation de la liste avec la premiere coordonnee.
17 liste2 = [] # Initialisation de la liste avec la deuxieme coordonnee.
18
19 liste3 = [] # Initialisation de la liste avec l'etiquette.
20
21 # Extraction des coordonnees destuples
22 for i in range(len(E)) :
23     liste1.append(E[i][0]) # liste1 contient les premieres coordonnees
24     liste2.append(E[i][1]) # liste2 contient les 2emes coordonnees
25     liste3.append(E[i][2]) # liste3 contient les etiquettes
26
27 # On affiche l'espace des caracteristiques (autrement dit des coordonnees de l'espace)
28 plt.scatter(liste1, liste2, marker = 'o', color = liste3) # Affiche les points
29
30 # Affiche le nom des axes
31 plt.xlabel('Age')
32 plt.ylabel("Nombre d'amis")
33
34 # Affiche les trois listes pour verification.
35 print("liste1 = ", liste1)
36 print("liste2 = ", liste2)
37 print("liste3 = ", liste3)
```

Utilisation de l'algorithme des k plus proches voisins pour déterminer quelle devrait être l'étiquette (couleur) d'un nouvel élément P.

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

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

```
Entrée [ ] : 1 import math # On utilisera math.sqrt pour le calcul de la distance.
2
```

```

3 def proches_voisins2(E, P, k, d):
4     """
5     Construit la liste des k plus proches voisins de x qui sont dans la list
6
7     Parametres nommes
8     -----
9     E : de type list
10         La liste des tuples (x_age, x_amis, couleur)
11
12     P : de type tuple
13         P = (xP, yP) le point dont on cherche les k plus proches voisins.
14
15     k : Le nombre de plus proches voisins qu'on souhaite trouver.
16
17     d : Fonction distance
18
19     Retourne
20     -----
21     voisins : de type list de tuples
22         La liste des k points plus proches voisins de x
23
24     """
25
26     voisins = []
27     for i in range(k): # On fait ici fonctionner l'algorithme avec k = 1.
28         voisins.append(E[i]) # Au depart, la liste voisins contient le premi
29
30     for i in range(k, len(E)): # On parcourt tout le reste de la liste E.
31         dist = d(P, E[i]) # On memorise la distance entre X et l'element E[i]
32         u = i # On memorise le rang i dans E de son element qui est a la dis
33         for j in range(k):
34             if j in range(k): # On parcourt toute la liste voisins.
35                 if dist < d(voisins[j], P): # Si la distance entre P et E[
36                     # l'element de rang j dans la
37                     dist = d(voisins[j], P) # alors on memorise la valeur d
38                     u = j # On memorise le rang j dans voisins de son eleme
39                 if u != i: # Si on a trouve un element dans E plus proche de P que
40                     voisins[u] = E[i] # alors E[i] vient remplacer cet element dans
41     return voisins
42
43
44 def d(x, y):
45     """
46     Calcule la distance euclidienne entre :
47     - le point P dont les coordonnees sont les deux premiers elements du tup
48     - le point Q dont les coordonnees sont les deux premiers elements du tup
49
50     Parametres nommes
51     -----
52     x, y : de type tuple
53         x(x[0], x[1], x[2]) donne les coordonnees du premier point P(x[0], x
54         y(y[0], y[1], y[2]) donne les coordonnees du deuxieme point Q(y[0],
55
56     Retourne
57     -----
58     dist : de type float
59         La distance euclidienne entre P et Q.
60
61     """
62     xP = x[0]
63     yP = x[1]
64     xQ = y[0]
65     yQ = y[1]
66
67     dist = math.sqrt((xQ - xP)**2 + (yQ - yP)**2)
68     return dist
69
70 k = 3
71 E = [[15, 1000, 'blue'], [20, 200, 'red'], [25, 300, 'red'], [30, 350, 'blue

```

```

72     [55, 400, 'red'], [60, 100, 'blue'], [65, 250, 'blue'], [70, 780, 'red']
73 P = (52, 142, 'green')
74 print("E = ", E)
75 print()

```

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

Réponse :

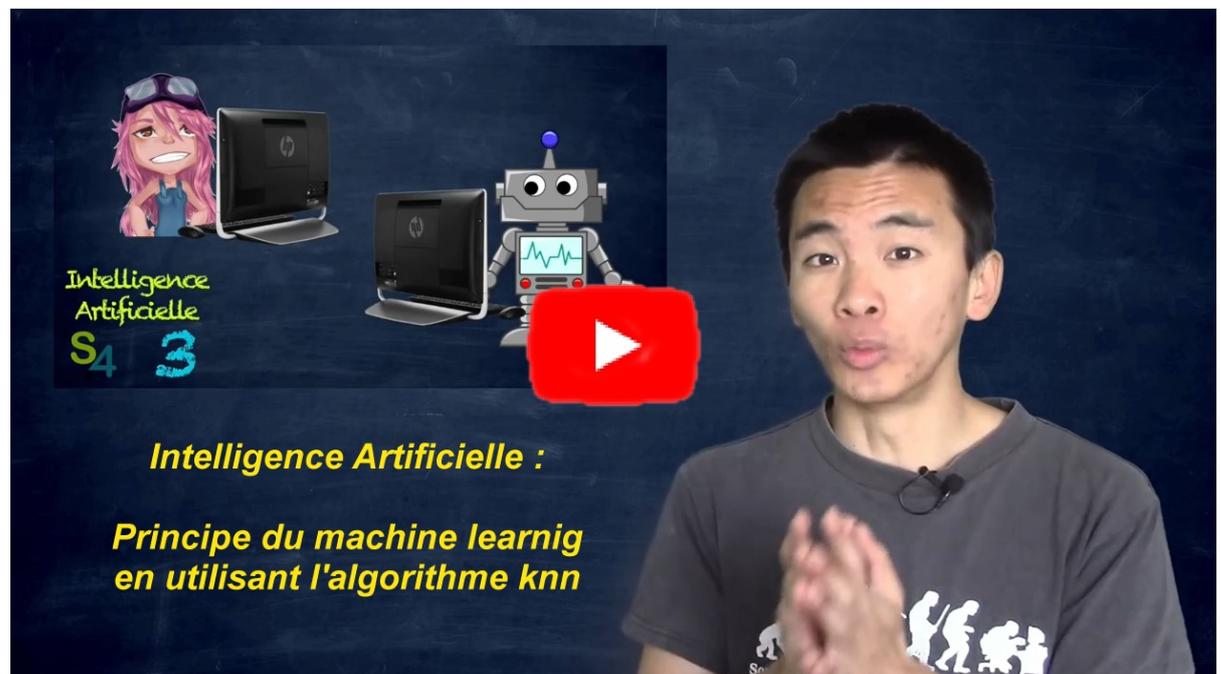
6) Quelle est la couleur de l'étiquette qu'on peut attribuer à P selon l'algorithme des k plus proches voisins lorsque k = 3 ? Justifiez votre réponse.

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



(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 reconnaître 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.

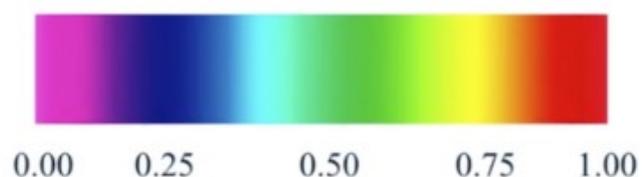
Les données se trouvent dans le fichier [donnees_fruits.csv](http://www.astrovirtuel.fr/jupyter/19_pnsi_cours/donnees_fruits.csv) (http://www.astrovirtuel.fr/jupyter/19_pnsi_cours/donnees_fruits.csv), à enregistrer dans le dossier où se trouve ce jupyter notebook.

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



Color category	color_score
Red	0.85 - 1.00
Orange	0.75 - 0.85
Yellow	0.65 - 0.75
Green	0.45 - 0.65

Le "score_couleur" est un flottant sur [0 ; 1] qui mesure la couleur. Vers 0 les couleurs "froides" d'un côté 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

Entrée []:

```

1 import csv
2 from collections import OrderedDict # structure de données utilisée
3
4 # On importe le fichier d'exemple de base de données de fruits
5 donnees_fruits_csv = open('donnees_fruits.csv', 'r', encoding='utf-8')
6 lecteur = csv.DictReader(donnees_fruits_csv, delimiter=';')
7 fruits = list(lecteur)
8
9 print("Identifiants des champs de données et type du champ")
10 items_fruits = list(fruits[0].items())
11 data = list(fruits[0].values())
12 for champ, donnee in items_fruits:
13     print("L'identifiant '" + champ, "' comporte des données de type ", type(donnee))
14     donnee)
15
16

```

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('')
```

```
Entrée [ ]: 1 ma_variable = "1.143"  
2 ma_variable_sans_guillemets = ma_variable.strip('')  
3  
4 print(ma_variable)  
5 print(ma_variable_sans_guillemets)
```

```
Entrée [ ]: 1 mon_flottant = float(ma_variable_sans_guillemets)  
2 print(mon_flottant)
```

```
Entrée [ ]: 1 # Pour importer une table avec Modification de type pour les champs contenu  
2 f = open("donnees_fruits.csv", "r", encoding = "utf-8")  
3 champs = f.readline() # Lecture de la première ligne  
4 lignes = f.readlines() # Lecture des autres lignes  
5 table=[]  
6  
7 champs2 = champs.rstrip().split(';') # Met les champs dans une liste  
8 print("champs2 avant = ", champs2)  
9  
10 champs2[0] = str(champs2[0]).strip('') # .strip('') retire le caractère  
11 champs2[1] = str(champs2[1]).strip('')  
12 champs2[2] = str(champs2[2]).strip('')  
13 champs2[3] = str(champs2[3]).strip('')  
14 champs2[4] = str(champs2[4]).strip('')  
15  
16  
17 for ligne in lignes:  
18     liste = ligne.rstrip().split(';')  
19     liste[0] = str(liste[0]).strip('')  
20     liste[1] = int(liste[1]) # Modification de domaine de valeur pour trans  
21     liste[2] = float(str(liste[2]).strip(''))  
22     liste[3] = float(str(liste[3]).strip(''))  
23     liste[4] = float(str(liste[4]).strip(''))  
24     table.append(liste)  
25 f.close()  
26  
27 print("champs2 après = ", champs2)  
28 print()
```

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, ...
```

```
Entrée [ ]: 1 from copy import deepcopy  
2
```

```

3 ma_copie = deepcopy(table)
4
5 fruits = [] # fruits sera la même liste de dictionnaires que fruit_texte,
6             # et l'index_couleur seront des flottants
7
8 # pour chaque ligne dans table
9 for ligne in table:
10
11     # Crée un dictionnaire
12     dico = {}
13
14     # remplit le dictionnaire
15     ...
16     ...
17     ...
18     ...
19     ...
20     fruits.append(dico)
21
22 print("fruits = ", fruits, "\n") # fruits_texte est une liste de dictionna
23 # dictionnaire représente comme un "p-upl

```

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.

```

Entrée [ ]: 1 sortes_de_fruits_au_total = [] # Contientra les sortes de fruits différent
2
3 for ligne in fruits:
4     ...
5     ...
6     ...
7

```

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*

```

Entrée [ ]: 1 import matplotlib.pyplot as plt
2 # import matplotlib.patches as mpatches
3 from matplotlib.lines import Line2D
4
5 #Données
6 nom = [fruit['nom_fruit'] for fruit in fruits]
7 code_couleur = []
8 code_marqueur = []
9 for i in range(len(nom)):
10     if nom[i] == "pomme":
11         code_couleur.append('green')
12         code_marqueur.append('.')
13     elif nom[i] == 'mandarine':
14         code_couleur.append('red')
15         code_marqueur.append('v')
16     elif nom[i] == 'orange':
17         code_couleur.append('orange')

```

```

18     code_marqueur.append('s')
19     else:
20         code_couleur.append('blue')
21         code_marqueur.append('*')
22 masses =[fruit['masse'] for fruit in fruits]
23 largeurs = [fruit['largeur'] for fruit in fruits]
24 hauteurs = [fruit['hauteur'] for fruit in fruits]
25 couleurs = [fruit['score_couleur'] for fruit in fruits]
26
27 #graphiques
28 #subplot permet de faire plusieurs graphiques (ici 3 lignes 2 colonnes, on p
29 fig, axes= plt.subplots(nrows=3, ncols=2,figsize=(18,18))
30
31 #pour la légende, compliquée à faire
32 legend_elements = [Line2D([0], [0], marker='o', color='w', label='Pommes', m
33                     Line2D([0], [0], marker='o', color='w', label='Mandarines
34                     Line2D([0], [0], marker='o', color='w', label='Oranges',
35                     Line2D([0], [0], marker='o', color='w', label='Citrons',
36                     ]
37
38 plt.subplot(321) # sous-schéma 1 dans les 3 colonnes et 2 lignes (3 - 2 -
39 plt.scatter(masses, largeurs, c = code_couleur, alpha = 0.5, s = 70)
40 plt.xlabel('masse')
41 plt.ylabel('largeur')
42 plt.legend(handles=legend_elements,loc = 'lower right')
43
44 """
45 # Autre méthode pour la légende, avec des rectangles
46 classes = ['pomme', 'mandarine', 'orange', 'citron']
47 class_colours = ['green','red','orange','yellow']
48 ronds = []
49 for i in range(0,len(class_colours)):
50     ronds.append(mpatches.Circle((0,0),5, color = class_colours[i])) # Circl
51 plt.legend(ronds,classes)
52 """
53
54 plt.subplot(322)
55 plt.scatter(masses, hauteurs, c = code_couleur, alpha = 0.5, s = 70)
56 plt.xlabel('masse')
57 plt.ylabel('hauteur')
58 plt.legend(handles=legend_elements,loc = 'lower right')
59
60 plt.subplot(323)
61 plt.scatter(masses, couleurs, c = code_couleur, alpha = 0.5, s = 70)
62 plt.xlabel('masse')
63 plt.ylabel('score couleur')
64 plt.legend(handles=legend_elements,loc = 'lower right')
65
66 plt.subplot(324)
67 plt.scatter(largeurs, hauteurs, c = code_couleur, alpha = 0.5, s = 70)
68 plt.xlabel('largeur')
69 plt.ylabel('hauteur')
70 plt.legend(handles=legend_elements,loc = 'lower right')
71
72 plt.subplot(325)
73 plt.scatter(largeurs, couleurs, c = code_couleur, alpha = 0.5, s = 70)
74 plt.xlabel('largeur')
75 plt.ylabel('score couleur')
76 plt.legend(handles=legend_elements,loc = 'lower left')
77
78 plt.subplot(326)
79 plt.scatter(hauteurs, couleurs, c = code_couleur, alpha = 0.5, s = 70)
80 plt.xlabel('hauteur')
81 plt.ylabel('score couleur')
82 plt.legend(handles=legend_elements,loc = 'lower left')
83
84

```

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 :

```
Entrée [ ]: 1 # Programme k-nn
2
3 from math import sqrt
4 from copy import deepcopy
5
6 def distance(pointA, pointB):
7     """
8     Renvoie la distance euclidienne entre deux points. Un point est vu comme
9
10
11     Parametres nommes :
12     -----
13     pointA : tuple de flottants. Ex : (192, 8.4, 7.3)
14
15     pointB : tuple de flottants, de même dimension que pointA. Ex : (180,
16
17
18     Retourne :
19     -----
20     distAB : flottant positif ou nul, distance euclidienne entre A et B. Ex
21
22     """
23
24
25
26
27     return distAB
28
29
30
31 def kPlusProches(point, liste, k):
32     """
```

```

33     Renvoie la liste des k plus proches voisins d'un point dans une liste de
34
35
36     Parametres nommes :
37     -----
38     point : tuple de flottants. Ex : (100, 6.3, 8.0)
39
40     liste : liste de tuples, les tuples ont comme dimension celle de point +
41             valeur de chaque tuple est la classe de l'individu, les autres s
42             coordonnées (valeurs des paramètres). Ex :
43             [('pomme', 192.0, 8.4, 7.3), ('pomme', 180.0, 8.0, 6.8), ...]
44
45     k :      entier strictement positif, inférieur ou égal au nombre d'élém
46
47
48     Retourne :
49     -----
50     k_voisins : liste de tuples, les tuples ont comme dimension celle de poi
51                 liste des voisins les plus proches de "point" dans "liste",
52                 la distance appelée dans cette fonction. Ex : [('pomme', 192
53                 ('mandarine', 76.0, 5.8, 4.0), ('pomme', 178.0, 7.1, 7.8)]
54
55     """
56
57
58
59
60     return k_voisins
61
62
63
64 def appartient_a(point, liste, k):
65     """
66     Renvoie la classe d'un individu déterminée par celle de ses k plus proch
67
68     Parametres nommes :
69     -----
70     point : tuple de flottants. Ex : (100, 6.3, 8.0)
71
72     liste : liste de tuples, les tuples ont comme dimension celle de point +
73             valeur de chaque tuple est la classe de l'individu, les autres s
74             coordonnées (valeurs des paramètres).
75
76     k :      entier strictement positif, inférieur ou égal au nombre d'élém
77
78
79     Retourne :
80     -----
81     classe : chaine de caractères donnant la classe de "point". C'est la cla
82             k-voisins les plus proches.
83
84     """
85
86
87
88
89     return classe
90
91
92
93
94
95
96 # Rassemblement des 3 paramètres en vue de l'exécution de la fonction
97 # appartient_a(point, liste, k)
98 # qui renvoie la classe majoritaire à laquelle apparinet vraisemblablement
99
100
101

```

```

102 fruit1 = (100, 6.3, 8.0)
103 fruit2 = (162, 7.1, 7.3)
104 mon_point = fruit2
105
106 # Fabrication de ma_liste = [('pomme', 192.0, 8.4, 7.3), ('pomme', 180.0, 8.0, 7.3), ...]
107 # à partir de la liste de dictionnaires
108 # fruits = [{'nom_fruit': 'pomme', 'masse': 192, 'largeur': 8.4, 'hauteur': 7.3}, ...]
109 #           {'nom_fruit': 'pomme', 'masse': 180, 'largeur': 8.0, ... }, ...]
110 ma_liste = []
111 for ligne in fruits:
112     ma_liste.append(tuple(ligne.values()))
113
114
115
116 nombre_de_voisins = 5
117
118
119
120 ma_classe = appartient_a(mon_point, ma_liste, nombre_de_voisins)
121 print("La classe du fruit est probablement :", ma_classe)
122
123

```

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).

Entrée []:

```

1  %matplotlib tk
2  import matplotlib.pyplot as plt
3  from mpl_toolkits.mplot3d import Axes3D
4
5  #graphiques
6  fig = plt.figure(figsize=(9,9))
7  ax = Axes3D(fig)
8
9  #pour la légende, compliquée à faire
10 legend_elements = [Line2D([0], [0], marker='o', color='w', label='Pommes', m
11                      Line2D([0], [0], marker='*', color='w', label='Mandarines
12                      Line2D([0], [0], marker='s', color='w', label='Oranges',
13                      Line2D([0], [0], marker='p', color='w', label='Citrons',
14                      Line2D([0], [0], marker='v', color='w', label='Inconnu',
15                          ]
16
17 #fig.add_subplot(111, projection = '3d')
18 for i in range(len(nom)):
19     if nom[i] == "pomme":
20         ax.scatter(masses[i], largeurs[i], hauteurs[i], c="green", s=70, alp
21     elif nom[i] == 'mandarine':
22         ax.scatter(masses[i], largeurs[i], hauteurs[i], c="red", s=70, marke
23     elif nom[i] == 'orange':
24         ax.scatter(masses[i], largeurs[i], hauteurs[i], c="orange", s=70, ma
25     else:
26         ax.scatter(masses[i], largeurs[i], hauteurs[i], c="blue", s=70, mark
27 #ax.scatter(masses, largeurs, hauteurs, c = code_couleur, s= 70)
28 fruit1 = [100, 6.3, 8.0]
29 fruit2 = [162, 7.1, 7.3]

```

```
30 ax.scatter(100, 6.3, 8.0, c='black', marker="v", s=110) # fruit1 est le pe
31 ax.scatter(162, 7.1, 7.3, c='black', marker="v", s=220) # fruit2 est le gr
32 ax.set_xlabel('masse')
33 ax.set_ylabel('largeur')
34 ax.set_zlabel('hauteur')
35 fig.legend(handles=legend_elements, loc = 'lower right')
36
37 plt.show()
```