TP1

May 11, 2020

0.1 Prise en main

[ ]:
```
x = 2434+33476
y = 1+3
```

[ ]:
```
x
```

[ ]:
```
print(x)
```

[ ]:
```
x
print(x)
```

[ ]:
```
y
```

[ ]:
```
y
print(x)
```

[ ]:
```
print(y)
x
```

[ ]:
```
print?
```

[ ]:
```
print
```

[5]:
```
import numpy as np
import scipy as sc
import matplotlib as mpl
import sklearn as skl
import time
```

1 Exercices

1.1 Exercice 1 : norme d'une liste

Ecrire une fonction Python qui calcule la norme Euclidienne d'un vecteur de dimension arbitraire définie sous forme de liste.

[6]:
```
def normeEuclidienne(vect):
    ret = 0
    for i in range(len(vect)):
        ret += vect[i] * vect[i]
    return np.sqrt(ret)
```

[7]:
```
vect = [12,8,4]
print("Norme du vecteur :",normeEuclidienne(vect))
```

```
Norme du vecteur : 14.966629547095765
```

Citez des avantages des listes en Python ? [A compléter]

1. Les listes sont dynamiques
2. On peut les remplir avec n'importe quoi
3. Elles sont simples à itérer

1.2 Exercice 1bis : matrice Numpy

- Ecrire une méthode créant une matrice, sous forme de listes de listes, de taille prédéfinie, remplie aléatoirement.
- Ecrire un produit matrice/vecteur utilisant cette structure et basée sur une double boucle.
- Comparer le temps d'exécution (package time) de votre produit avec la même opération réalisée par Numpy.

[4]:
```
# Import de package
import random  # https://docs.python.org/3.7/library/random.html
import time    # https://docs.python.org/3.7/library/time.html
```

[2]:
```
def createMatRandom(low,high,sizeX,sizeY):
    return np.random.randint(low,high,(sizeX,sizeY))
def matVectProduct(mat,vect):
    ret = np.zeros(mat.shape[0])
    if(mat.shape[1] == len(vect)):
        for i in range(mat.shape[0]):
            sum = 0
            for j in range(mat.shape[1]):
                sum += mat[i][j] * vect[j]
            ret[i] = sum
        return ret
    else:
        return False
```

[10]:
```
mat = createMatRandom(0,6,10,10)
vect = np.random.randint(0,9,10)
print(mat)
print(vect)
t1 = time.clock()
print("Product :",matVectProduct(mat,vect))
t2 = time.clock()
print("Time :",(t2-t1),"s")
t1 = time.clock()
print("Numpy prod :",mat.dot(vect))
t2 = time.clock()
print("Time numpy :",(t2-t1),"s")
```

```
[[5 4 0 5 3 0 4 3 1 0]
 [4 4 1 5 5 5 5 0 0 1]
 [1 5 3 2 1 4 2 1 1 3]
 [1 0 0 1 0 2 1 3 5 2]
 [1 2 4 5 1 2 4 3 0 5]
 [0 3 3 4 0 5 0 4 4 5]
 [2 0 1 4 2 2 2 5 3 4]
 [2 2 4 0 5 2 3 4 0 2]
 [4 5 2 1 3 3 2 3 4 3]
 [0 3 5 5 2 5 0 5 0 5]]
[6 3 6 7 4 6 4 2 1 5]
Product : [112. 152. 107.  50. 134. 122. 107. 104. 121. 147.]
Time : 0.0006167999999888707 s
Numpy prod : [112 152 107  50 134 122 107 104 121 147]
Time numpy : 0.00030950000000018463 s

C:\Users\Megaport\Anaconda3\lib\site-packages\ipykernel_launcher.py:5:
DeprecationWarning: time.clock has been deprecated in Python 3.3 and will be
removed from Python 3.8: use time.perf_counter or time.process_time instead
  """
C:\Users\Megaport\Anaconda3\lib\site-packages\ipykernel_launcher.py:7:
DeprecationWarning: time.clock has been deprecated in Python 3.3 and will be
removed from Python 3.8: use time.perf_counter or time.process_time instead
  import sys
C:\Users\Megaport\Anaconda3\lib\site-packages\ipykernel_launcher.py:9:
DeprecationWarning: time.clock has been deprecated in Python 3.3 and will be
removed from Python 3.8: use time.perf_counter or time.process_time instead
  if __name__ == '__main__':
C:\Users\Megaport\Anaconda3\lib\site-packages\ipykernel_launcher.py:11:
DeprecationWarning: time.clock has been deprecated in Python 3.3 and will be
removed from Python 3.8: use time.perf_counter or time.process_time instead
  # This is added back by InteractiveShellApp.init_path()
```

Quel temps obtenez vous ? [A compléter]

Cas | (m, n) | **Boucle for** | **Solution Numpy** :-: | :-: | :-:

Cas 1 |(3,3) |0.0003 s |0.0002 s Cas 2 |(100,100) |0.004 s |0.0002 s Cas 3 |(1000,1000) |0.3 s |0.002 s

1.3 Exercice 2 : pickle

- Définissez votre propre liste de chaînes de caractère (par exemple, une liste de villes de Bretagne).
- Utilisez le module `pickle` pour sauver cette liste dans un fichier texte : Pickle.
- Rechargez cette liste.
- Même chose avec un dictionnaire.

[48]:
```
import pickle
villes = ["Vannes","Quimper","Lorient","Brest","Rennes","Saint-Malo"]
pickle.dump(villes, open('villes.txt', 'wb'))
villesLoad = pickle.load(open('villes.txt', 'rb'))
print(villesLoad)
```

```
['Vannes', 'Quimper', 'Lorient', 'Brest', 'Rennes', 'Saint-Malo']
```

[49]:
```
villesDict = {
    "name1" : "Vannes",
    "name2" : "Quimper",
    "name3" : "Lorient",
    "name4" : "Brest",
    "name5" : "Rennes",
    "name6" : "Saint-Malo",
}
np.save("villes.npy", villesDict)
villesDictLoad = np.load("villes.npy", allow_pickle="True")
print(villesDictLoad.item())
```

```
{'name1': 'Vannes', 'name2': 'Quimper', 'name3': 'Lorient', 'name4': 'Brest',
'name5': 'Rennes', 'name6': 'Saint-Malo'}
```

1.4 Exercice 3 : Classe

- Créer une classe Python nommée `Vecteur`, que l'on utilisera pour décrire un vecteur plan. Ces paramètres sont `x` et `y`.
- Implémenter l'addition `__add__`, la soustraction `__sub__`, les multiplications `__mul__` et `__rmul__` par un vecteur et un scalaire, et la représentation `__repr__`. Ne pas oublier le constructeur `__init__`.
- Créer une classe nommée `Particule`, dont les paramètres sont les vecteurs position et vitesse. Implémenter une méthode `deplace(dt)` pour calculer la nouvelle position d'une particule après un pas de temps dt.

[50]:
```
class Vecteur:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __add__(self,otherVect):
        return Vecteur(self.x+otherVect.x, self.y+otherVect.y)
    def __sub__(self,otherVect):
        return Vecteur(self.x-otherVect.x, self.y-otherVect.y)
    def __mul__(self,otherVect):
        return Vecteur(self.x*otherVect.x, self.y*otherVect.y)
    def __rmul__(self,scalaire):
        return Vecteur(self.x*scalaire, self.y*scalaire)
    def __repr__(self):
```

```
        return "("+str(self.x)+","+str(self.y)+")"
```

[51]:
```
class Particule:
    def __init__(self,vectPos,vectVit):
        self.vectPos = vectPos
        self.vectVit = vectVit
    def deplace(self,dt):
        self.vectPos = self.vectPos + (dt * self.vectVit)
```

Testez le bon fonctionnement de vos deux classes

[52]:
```
vect1 = Vecteur(4,5)
vect2 = Vecteur(6,7)
print("Vecteur 1 :",vect1)
print("Vecteur 2 :",vect2)
print("Addition :",(vect1 + vect2))
print("Soustraction :",(vect1 - vect2))
print("Produit vectorielle :",(vect1 * vect2))
scalaire = 2
print("Produit de Vecteur1 par",scalaire,":",(scalaire*vect1))

particule = Particule(vect1,vect2)
print("Particule, vectPos :",particule.vectPos,", vectVit :",particule.vectVit)
dt = 3
particule.deplace(dt)
print("Deplacement avec dt =",dt,": vectPos =",particule.vectPos)
```

```
Vecteur 1 : (4,5)
Vecteur 2 : (6,7)
Addition : (10,12)
Soustraction : (-2,-2)
Produit vectorielle : (24,35)
Produit de Vecteur1 par 2 : (8,10)
Particule, vectPos : (4,5) , vectVit : (6,7)
Deplacement avec dt = 3 : vectPos = (22,26)
```

1.5 Exercice 4 : Sympy

$$I = \int_{-\infty}^{+\infty} e^{-x^2} dx$$

- Calculez l'intégrale suivante (dite de Gauss) en utilisant la méthode symbolique de Sympy
- Calculez cette même intégrale numériquement en utilisant la méthode des trapèzes de numpy (i.e. `numpy.trapz`)
- Calculez une nouvelle fois la même intégrale avec une méthode de calcul équivalente que vous aurez codée.

[53]:
```
from sympy import *
from mpmath import inf
import numpy as np
```

[54]:
```
x = Symbol('x')
print("Avec Sympy :")
integrate(exp(-(x**2)),(x,-inf,+inf))
#quad(lambda x: exp(-(x**2)), [-inf, +inf])
```

```
Avec Sympy :
```

[54]: $1.0\sqrt{\pi}$

[55]:
```
def gaussian(x):
    return np.exp(-(x**2))
x = np.arange(-10,10,0.1)
y = gaussian(x)
print("Avec numpy.trapz de -10 à 10 :",np.trapz(y,x))
#Environ sqrt(PI)
```

```
Avec numpy.trapz de -10 à 10 : 1.7724538509055159
```

[56]:
```
def integrale(f,borneInf,borneSup,pas):
    x = np.arange(borneInf,borneSup,pas)
    y = f(x)
    ret = 0
    for i in range(len(y)):
        ret += y[i]
    return ret*pas
print("Avec ma fonction de -10 à 10 :",integrale(gaussian,-10,10,0.1))
```

```
Avec ma fonction de -10 à 10 : 1.7724538509055237
```

1.6 Exercice 5 : image

- Récupérez l'image hyperspectrale nommée `Urban_R162.mat`
- Examinez la structure de données correspondantes
- Faire une fonction qui affiche une bande spectrale donnée de l'image sous la forme d'une image avec une colormap
- Affichez une composition couleur de cette image en choisissant la correspondance spectre/couleur suivante :

50->R 100->G 150->B * Affichez sur un même diagramme 10 spectres pris aléatoirement dans l'image

[57]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from matplotlib import colors
import math
```

[58]:
```
# Lecture d'un fichier .mat en Python
import scipy.io
mat = scipy.io.loadmat('Urban_R162.mat')
```

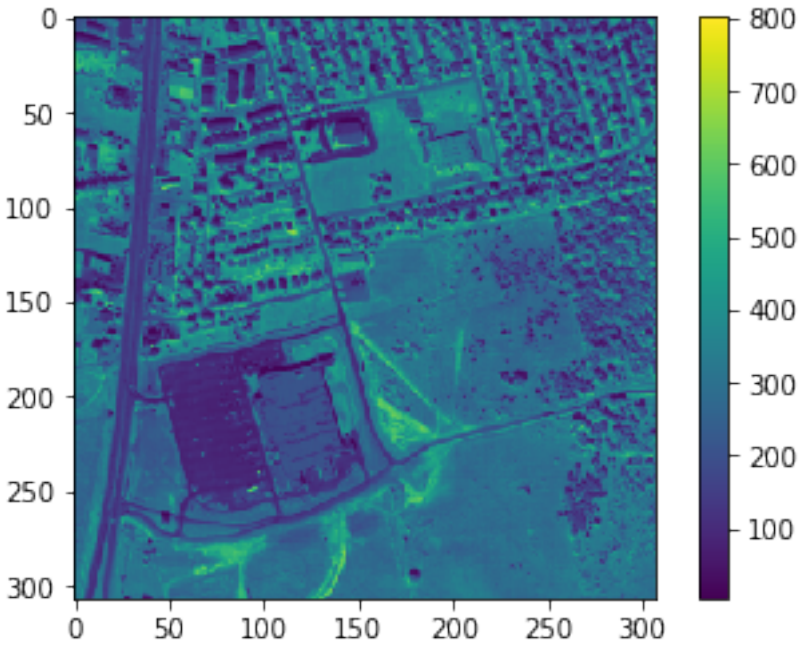
[59]:
```
print(mat.keys())
print(mat['SlectBands'].size)
print(mat['nRow'], mat['nCol'], mat['nBand'])
print(mat['maxValue'])
print(mat['Y'].shape)
```

```
dict_keys(['__header__', '__version__', '__globals__', 'SlectBands', 'nRow',
'nCol', 'nBand', 'Y', 'maxValue'])
162
[[307]] [[307]] [[210]]
[[1000]]
(162, 94249)
```

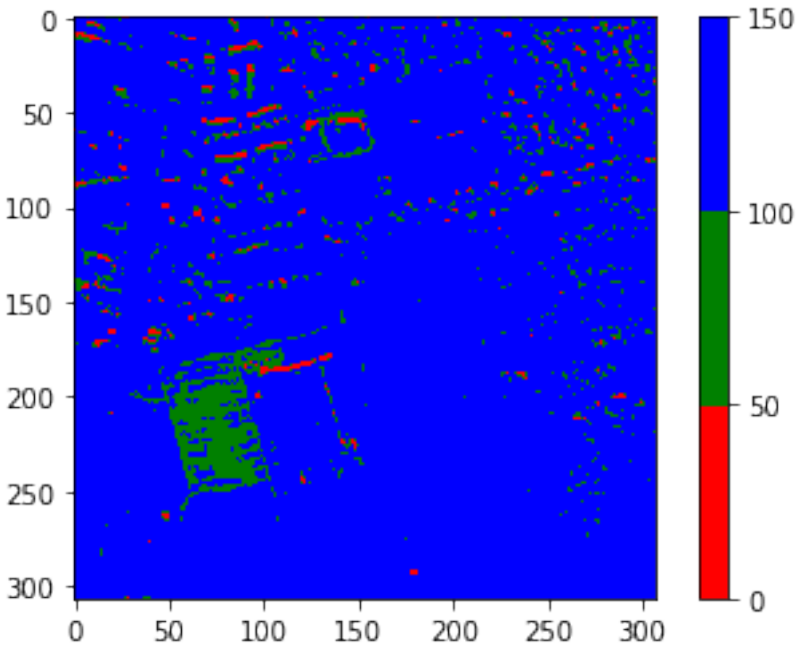
Comment est organisé la donnée (quelle structure ?) ? Quelle est la taille de l'image ? [A compléter]

La structure de données est organisée sous forme de dictionnaire. Les clés nRow et nCol définisse la taille de l'image qui est donc de 307x307. Dans Y on a l'image (94249 pixels) qui est définie pour chaque bande spectrale sélectionnée (162). Dans SlectBands on nBand bandes spectrales disponibles pour cette image. L'attribut maxValue correspond à la valeur maximale se trouvant dans l'image.

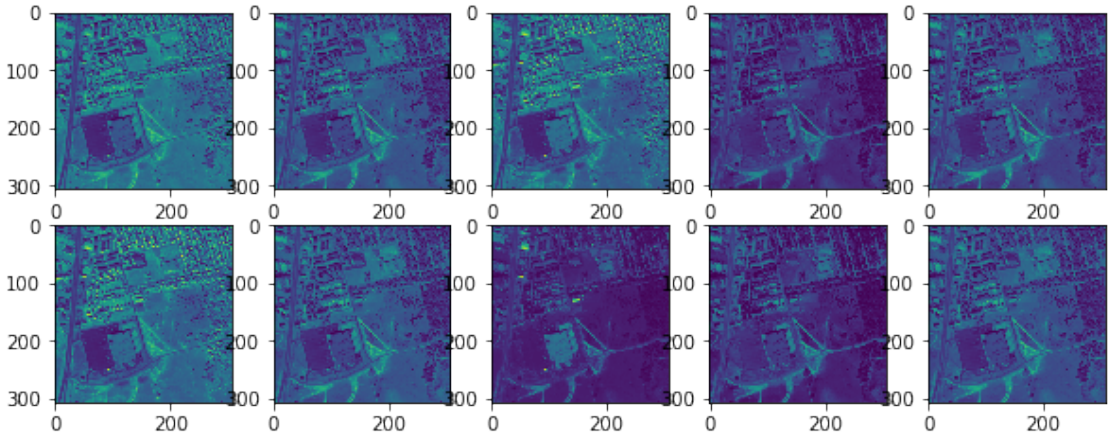
[60]:
```
def getSpecBand(mat,band):
    bands = mat['SlectBands'].flatten().tolist()
    if band in bands:
        ind = bands.index(band)
        return mat['Y'][ind].reshape((mat['nRow'][0][0],mat['nCol'][0][0]))
    return False

def displaySpecBand(mat,band):
    res = getSpecBand(mat,band)
    if type(res) != bool:
        plt.imshow(res)
        plt.colorbar()
    else:
        print("This band is not available")

displaySpecBand(mat,100)
```



[61]:
```
cmap = colors.ListedColormap(['red', 'green', 'blue'])
bounds = [0,50,100,150]

def displaySpecBandCmap(mat,band,cmap,bounds):
    res = getSpecBand(mat,band)
    norm = colors.BoundaryNorm(bounds, cmap.N)
    if type(res) != bool:
        plt.imshow(res,cmap=cmap, norm=norm)
        plt.colorbar()
    else:
        print("This band is not available")

displaySpecBandCmap(mat,100,cmap,bounds)
```



[62]:
```
def displayRandomBands(mat,n):
    minVal = mat['SlectBands'][0][0]
    maxVal = mat['SlectBands'][-1][0]
    i = 0
    size = math.floor(n/2)
    fig = plt.figure(figsize=(10, 10))
    while(i < n):
        band = np.random.randint(minVal,maxVal)
        res = getSpecBand(mat,band)
        if type(res) != bool:
            fig.add_subplot(size, size, i+1)
            im = plt.imshow(res)
            i = i+1
    plt.show()
displayRandomBands(mat,10)
```


0
100
200
300
0
200

TP_ACP

May 11, 2020

1 Analyse en Composantes Principales (ACP)

1.1 Utilisation de Pandas et de ses dataframes

1.1.1 Lecture du fichier

- Lire le fichier `avocado.csv` dans un dataframe avec pandas
- Afficher le dataframe
- A quoi correspondent les colonnes `4046`, `4225`, `4770` ?

Source : https://www.kaggle.com/neuromusic/avocado-prices (jeu de données extrait de Kaggle)

[30]:
```
from pandas import DataFrame, read_csv
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from numpy.random import randint as rand

avocado = pd.read_csv('avocado.csv')
print(avocado)
```

```
       Unnamed: 0        Date  AveragePrice  Total Volume       4046  \
0               0  2015-12-27          1.33      64236.62    1036.74
1               1  2015-12-20          1.35      54876.98     674.28
2               2  2015-12-13          0.93     118220.22     794.70
3               3  2015-12-06          1.08      78992.15    1132.00
4               4  2015-11-29          1.28      51039.60     941.48
5               5  2015-11-22          1.26      55979.78    1184.27
6               6  2015-11-15          0.99      83453.76    1368.92
7               7  2015-11-08          0.98     109428.33     703.75
8               8  2015-11-01          1.02      99811.42    1022.15
9               9  2015-10-25          1.07      74338.76     842.40
10             10  2015-10-18          1.12      84843.44     924.86
11             11  2015-10-11          1.28      64489.17    1582.03
12             12  2015-10-04          1.31      61007.10    2268.32
13             13  2015-09-27          0.99     106803.39    1204.88
14             14  2015-09-20          1.33      69759.01    1028.03
15             15  2015-09-13          1.28      76111.27     985.73
```

16	16	2015-09-06	1.11	99172.96	879.45
17	17	2015-08-30	1.07	105693.84	689.01
18	18	2015-08-23	1.34	79992.09	733.16
19	19	2015-08-16	1.33	80043.78	539.65
20	20	2015-08-09	1.12	111140.93	584.63
21	21	2015-08-02	1.45	75133.10	509.94
22	22	2015-07-26	1.11	106757.10	648.75
23	23	2015-07-19	1.26	96617.00	1042.10
24	24	2015-07-12	1.05	124055.31	672.25
25	25	2015-07-05	1.35	109252.12	869.45
26	26	2015-06-28	1.37	89534.81	664.23
27	27	2015-06-21	1.27	104849.39	804.01
28	28	2015-06-14	1.32	89631.30	850.58
29	29	2015-06-07	1.07	122743.06	656.71
...	...	...	...	...	...
18219	6	2018-02-11	1.56	1317000.47	98465.26
18220	7	2018-02-04	1.53	1384683.41	117922.52
18221	8	2018-01-28	1.61	1336979.09	118616.17
18222	9	2018-01-21	1.63	1283987.65	108705.28
18223	10	2018-01-14	1.59	1476651.08	145680.62
18224	11	2018-01-07	1.51	1517332.70	129541.43
18225	0	2018-03-25	1.60	271723.08	26996.28
18226	1	2018-03-18	1.73	210067.47	33437.98
18227	2	2018-03-11	1.63	264691.87	27566.25
18228	3	2018-03-04	1.46	347373.17	25990.60
18229	4	2018-02-25	1.49	301985.61	34200.18
18230	5	2018-02-18	1.64	224798.60	30149.00
18231	6	2018-02-11	1.47	275248.53	24732.55
18232	7	2018-02-04	1.41	283378.47	22474.66
18233	8	2018-01-28	1.80	185974.53	22918.40
18234	9	2018-01-21	1.83	189317.99	27049.44
18235	10	2018-01-14	1.82	207999.67	33869.12
18236	11	2018-01-07	1.48	297190.60	34734.97
18237	0	2018-03-25	1.62	15303.40	2325.30
18238	1	2018-03-18	1.56	15896.38	2055.35
18239	2	2018-03-11	1.56	22128.42	2162.67
18240	3	2018-03-04	1.54	17393.30	1832.24
18241	4	2018-02-25	1.57	18421.24	1974.26
18242	5	2018-02-18	1.56	17597.12	1892.05
18243	6	2018-02-11	1.57	15986.17	1924.28
18244	7	2018-02-04	1.63	17074.83	2046.96
18245	8	2018-01-28	1.71	13888.04	1191.70
18246	9	2018-01-21	1.87	13766.76	1191.92
18247	10	2018-01-14	1.93	16205.22	1527.63
18248	11	2018-01-07	1.62	17489.58	2894.77

						\
	4225	4770	Total Bags	Small Bags	Large Bags	XLarge Bags
0	54454.85	48.16	8696.87	8603.62	93.25	0.00

1	44638.81	58.33	9505.56	9408.07	97.49	0.00
2	109149.67	130.50	8145.35	8042.21	103.14	0.00
3	71976.41	72.58	5811.16	5677.40	133.76	0.00
4	43838.39	75.78	6183.95	5986.26	197.69	0.00
5	48067.99	43.61	6683.91	6556.47	127.44	0.00
6	73672.72	93.26	8318.86	8196.81	122.05	0.00
7	101815.36	80.00	6829.22	6266.85	562.37	0.00
8	87315.57	85.34	11388.36	11104.53	283.83	0.00
9	64757.44	113.00	8625.92	8061.47	564.45	0.00
10	75595.85	117.07	8205.66	7877.86	327.80	0.00
11	52677.92	105.32	10123.90	9866.27	257.63	0.00
12	49880.67	101.36	8756.75	8379.98	376.77	0.00
13	99409.21	154.84	6034.46	5888.87	145.59	0.00
14	59313.12	150.50	9267.36	8489.10	778.26	0.00
15	65696.86	142.00	9286.68	8665.19	621.49	0.00
16	90062.62	240.79	7990.10	7762.87	227.23	0.00
17	94362.67	335.43	10306.73	10218.93	87.80	0.00
18	67933.79	444.78	10880.36	10745.79	134.57	0.00
19	68666.01	394.90	10443.22	10297.68	145.54	0.00
20	100961.46	368.95	9225.89	9116.34	109.55	0.00
21	62035.06	741.08	11847.02	11768.52	78.50	0.00
22	91949.05	966.61	13192.69	13061.53	131.16	0.00
23	82049.40	2238.02	11287.48	11103.49	183.99	0.00
24	94693.52	4257.64	24431.90	24290.08	108.49	33.33
25	72600.55	5883.16	29898.96	29663.19	235.77	0.00
26	57545.79	4662.71	26662.08	26311.76	350.32	0.00
27	76688.55	5481.18	21875.65	21662.00	213.65	0.00
28	55400.94	4377.19	29002.59	28343.14	659.45	0.00
29	99220.82	90.32	22775.21	22314.99	460.22	0.00
...	...	...	...	...	...	...
18219	270798.27	1839.80	945638.02	768242.42	177144.00	251.60
18220	287724.61	1703.52	977084.84	774695.74	201878.69	510.41
18221	280080.34	1270.61	936859.49	796104.27	140652.84	102.38
18222	259172.13	1490.02	914409.26	710654.40	203526.59	228.27
18223	323669.83	1580.01	1005593.78	858772.69	146808.97	12.12
18224	296490.29	1289.07	1089861.24	915452.78	174381.57	26.89
18225	77861.39	117.56	166747.85	87108.00	79495.39	144.46
18226	47165.54	110.40	129353.55	73163.12	56020.24	170.19
18227	60383.57	276.42	176465.63	107174.93	69290.70	0.00
18228	71213.19	79.01	250090.37	85835.17	164087.33	167.87
18229	49139.34	85.58	218560.51	99989.62	118314.77	256.12
18230	38800.64	123.13	155725.83	120428.13	35257.73	39.97
18231	61713.53	243.00	188559.45	88497.05	99810.80	251.60
18232	55360.49	133.41	205409.91	70232.59	134666.91	510.41
18233	33051.14	93.52	129911.47	77822.23	51986.86	102.38
18234	33561.32	439.47	128267.76	76091.99	51947.50	228.27
18235	47435.14	433.52	126261.89	89115.78	37133.99	12.12
18236	62967.74	157.77	199330.12	103761.55	95544.39	24.18

18237	2171.66	0.00	10806.44	10569.80	236.64	0.00
18238	1499.55	0.00	12341.48	12114.81	226.67	0.00
18239	3194.25	8.93	16762.57	16510.32	252.25	0.00
18240	1905.57	0.00	13655.49	13401.93	253.56	0.00
18241	2482.65	0.00	13964.33	13698.27	266.06	0.00
18242	1928.36	0.00	13776.71	13553.53	223.18	0.00
18243	1368.32	0.00	12693.57	12437.35	256.22	0.00
18244	1529.20	0.00	13498.67	13066.82	431.85	0.00
18245	3431.50	0.00	9264.84	8940.04	324.80	0.00
18246	2452.79	727.94	9394.11	9351.80	42.31	0.00
18247	2981.04	727.01	10969.54	10919.54	50.00	0.00
18248	2356.13	224.53	12014.15	11988.14	26.01	0.00

	type	year	region
0	conventional	2015	Albany
1	conventional	2015	Albany
2	conventional	2015	Albany
3	conventional	2015	Albany
4	conventional	2015	Albany
5	conventional	2015	Albany
6	conventional	2015	Albany
7	conventional	2015	Albany
8	conventional	2015	Albany
9	conventional	2015	Albany
10	conventional	2015	Albany
11	conventional	2015	Albany
12	conventional	2015	Albany
13	conventional	2015	Albany
14	conventional	2015	Albany
15	conventional	2015	Albany
16	conventional	2015	Albany
17	conventional	2015	Albany
18	conventional	2015	Albany
19	conventional	2015	Albany
20	conventional	2015	Albany
21	conventional	2015	Albany
22	conventional	2015	Albany
23	conventional	2015	Albany
24	conventional	2015	Albany
25	conventional	2015	Albany
26	conventional	2015	Albany
27	conventional	2015	Albany
28	conventional	2015	Albany
29	conventional	2015	Albany
...	...	...	...
18219	organic	2018	TotalUS
18220	organic	2018	TotalUS
18221	organic	2018	TotalUS

```
18222       organic  2018           TotalUS
18223       organic  2018           TotalUS
18224       organic  2018           TotalUS
18225       organic  2018              West
18226       organic  2018              West
18227       organic  2018              West
18228       organic  2018              West
18229       organic  2018              West
18230       organic  2018              West
18231       organic  2018              West
18232       organic  2018              West
18233       organic  2018              West
18234       organic  2018              West
18235       organic  2018              West
18236       organic  2018              West
18237       organic  2018  WestTexNewMexico
18238       organic  2018  WestTexNewMexico
18239       organic  2018  WestTexNewMexico
18240       organic  2018  WestTexNewMexico
18241       organic  2018  WestTexNewMexico
18242       organic  2018  WestTexNewMexico
18243       organic  2018  WestTexNewMexico
18244       organic  2018  WestTexNewMexico
18245       organic  2018  WestTexNewMexico
18246       organic  2018  WestTexNewMexico
18247       organic  2018  WestTexNewMexico
18248       organic  2018  WestTexNewMexico

[18249 rows x 14 columns]
```

Les colonnes 4046, 4225, 4770 correspondent respectivement aux nombres d'avocats vendus avec leur code PLU.

1.1.2 Utilisation du dataframe

Tester les fonctions suivantes sur le dataframe (`Shift+Tab` pour obtenir la documentation) :

- `columns`
- `head(n)`, `tail(n)`. Quel est le rôle de `n` ?
- `shape`
- `dtypes`
- `to_numpy()`, `values` et `index`
- `describe()`
- `mean()`
- `loc[]` et `iloc[]`

[8]:
```
print(avocado.columns)
```

```
Index(['Unnamed: 0', 'Date', 'AveragePrice', 'Total Volume', '4046', '4225',
       '4770', 'Total Bags', 'Small Bags', 'Large Bags', 'XLarge Bags', 'type',
       'year', 'region'],
      dtype='object')
```

head(n) affiche le n premier éléments, tail de n affiche les n derniers éléments.

[11]:
```
print(avocado.shape)
```

```
(18249, 14)
```

[12]:
```
print(avocado.dtypes)
```

```
Unnamed: 0        int64
Date             object
AveragePrice    float64
Total Volume    float64
4046            float64
4225            float64
4770            float64
Total Bags      float64
Small Bags      float64
Large Bags      float64
XLarge Bags     float64
type             object
year              int64
region           object
dtype: object
```

[15]:
```
print(avocado.to_numpy())
print(avocado.values)
print(avocado.index)
```

```
[[0 '2015-12-27' 1.33 ... 'conventional' 2015 'Albany']
 [1 '2015-12-20' 1.35 ... 'conventional' 2015 'Albany']
 [2 '2015-12-13' 0.93 ... 'conventional' 2015 'Albany']
 ...
 [9 '2018-01-21' 1.87 ... 'organic' 2018 'WestTexNewMexico']
 [10 '2018-01-14' 1.93 ... 'organic' 2018 'WestTexNewMexico']
 [11 '2018-01-07' 1.62 ... 'organic' 2018 'WestTexNewMexico']]
[[0 '2015-12-27' 1.33 ... 'conventional' 2015 'Albany']
 [1 '2015-12-20' 1.35 ... 'conventional' 2015 'Albany']
 [2 '2015-12-13' 0.93 ... 'conventional' 2015 'Albany']
 ...
 [9 '2018-01-21' 1.87 ... 'organic' 2018 'WestTexNewMexico']
 [10 '2018-01-14' 1.93 ... 'organic' 2018 'WestTexNewMexico']
 [11 '2018-01-07' 1.62 ... 'organic' 2018 'WestTexNewMexico']]
RangeIndex(start=0, stop=18249, step=1)
```

[16]:
```
print(avocado.describe())
```

```
         Unnamed: 0  AveragePrice  Total Volume          4046          4225  \
count  18249.000000  18249.000000  1.824900e+04  1.824900e+04  1.824900e+04
mean      24.232232      1.405978  8.506440e+05  2.930084e+05  2.951546e+05
std       15.481045      0.402677  3.453545e+06  1.264989e+06  1.204120e+06
min        0.000000      0.440000  8.456000e+01  0.000000e+00  0.000000e+00
25%       10.000000      1.100000  1.083858e+04  8.540700e+02  3.008780e+03
50%       24.000000      1.370000  1.073768e+05  8.645300e+03  2.906102e+04
75%       38.000000      1.660000  4.329623e+05  1.110202e+05  1.502069e+05
max       52.000000      3.250000  6.250565e+07  2.274362e+07  2.047057e+07

               4770    Total Bags    Small Bags    Large Bags    XLarge Bags  \
count  1.824900e+04  1.824900e+04  1.824900e+04  1.824900e+04   18249.000000
mean   2.283974e+04  2.396392e+05  1.821947e+05  5.433809e+04    3106.426507
std    1.074641e+05  9.862424e+05  7.461785e+05  2.439660e+05   17692.894652
min    0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00       0.000000
25%    0.000000e+00  5.088640e+03  2.849420e+03  1.274700e+02       0.000000
50%    1.849900e+02  3.974383e+04  2.636282e+04  2.647710e+03       0.000000
75%    6.243420e+03  1.107834e+05  8.333767e+04  2.202925e+04     132.500000
max    2.546439e+06  1.937313e+07  1.338459e+07  5.719097e+06  551693.650000

              year
count  18249.000000
mean    2016.147899
std        0.939938
min     2015.000000
25%     2015.000000
50%     2016.000000
75%     2017.000000
max     2018.000000
```

[17]:
```
print(avocado.mean())
```

```
Unnamed: 0          24.232232
AveragePrice         1.405978
Total Volume    850644.013009
4046            293008.424531
4225            295154.568356
4770             22839.735993
Total Bags      239639.202060
Small Bags      182194.686696
Large Bags       54338.088145
XLarge Bags       3106.426507
year              2016.147899
dtype: float64
```

[23]:
```
avocado.loc[100]
```

[23]:
```
Unnamed: 0                48
Date              2015-01-25
AveragePrice             1.1
Total Volume          449333
4046                  393408
4225                 18718.3
4770                  594.25
Total Bags           36612.2
Small Bags           13176.4
Large Bags           23435.8
XLarge Bags                0
type            conventional
year                    2015
region               Atlanta
Name: 100, dtype: object
```

[24]:
```
avocado.iloc[100]
```

[24]:
```
Unnamed: 0                48
Date              2015-01-25
AveragePrice             1.1
Total Volume          449333
4046                  393408
4225                 18718.3
4770                  594.25
Total Bags           36612.2
Small Bags           13176.4
Large Bags           23435.8
XLarge Bags                0
type            conventional
year                    2015
region               Atlanta
Name: 100, dtype: object
```

2 Mise en pratique

Répondre aux questions suivantes : [A COMPLETER]

1. Quelle est la taille du dataframe (nombre de colonnes, nombre de lignes) ?
2. Combien y a-t-il de types d'avocat ? Lesquels ?

[91]:
```
print("Nombre de lignes :",avocado.shape[0],"/ Nombre de colonnes :",avocado.
 ↪shape[1])
print(avocado)
```

```
Nombre de lignes : 18249 / Nombre de colonnes : 14
255486
```

[93]:
```
print(avocado['type'].unique())
```

```
['conventional' 'organic']
```

Créer un dataframe `data` qui contient 50 échantillons d'avocats `conventional` et 50 échantillons d'avocats `organic` tirés aléatoirement des données `avocado.csv`. On gardera les informations suivantes : * `AveragePrice` * `Total Volume` * 4046 * 4225 * 4770 * `Total Bags` * `Small Bags` * `Large Bags` * `XLarge Bags`

Vérifier que `data` comporte 100 lignes et 10 colonnes.

[115]:
```
col = ['AveragePrice', 'Total Volume', '4046', '4225', '4770',
       'Total Bags', 'Small Bags', 'Large Bags', 'XLarge Bags', 'type', 'year',
       'region']
conv = avocado[avocado['type']=='conventional'].sample(50)
org = avocado[avocado['type']=='organic'].sample(50)
data = pd.concat([conv, org]).loc[:,col]
print(data)
```

```
      AveragePrice  Total Volume        4046        4225       4770  \
4006          0.79     105228.60     1367.94    44377.90     142.37
2449          1.12      56013.43    17535.51    21806.06    2671.75
889           1.27     212091.14     3720.08   176358.45     319.25
797           1.04    3388803.79   478858.03  1877733.21  419390.50
992           1.08     108444.41     3079.44    65254.08   10854.59
7986          1.38     345869.64   153802.62    72253.48    2212.93
3524          1.09     371639.10    78492.11    67761.91  111819.43
7190          1.65    4043204.76   222626.36  2576783.08   14146.92
5516          1.09    4187550.98  1525325.69   938768.71   86789.70
6173          0.86     243935.99   100495.93    26548.19    1160.37
6968          1.27    3147519.14   673327.18  1319522.05   27318.06
2376          0.82    5593885.09  3657278.94  1183497.85  114923.71
7007          1.70     154956.62    73759.23    21324.74      90.32
589           0.82    1101680.91   674670.89   309275.19    7468.25
2632          1.18     275260.08   182281.62    18249.86      79.99
1186          0.86    2797745.89  2005781.71   400644.42   62567.60
7391          0.77    1224659.64   551528.30   185311.58   14893.87
1919          0.98     776237.93   165259.75   535324.57   15903.81
4530          0.94     911367.38   437810.54   189026.34   15106.03
2728          1.01    5791508.96  2688406.31  1950436.22  133097.00
2247          0.90     640331.58    87081.69   301656.56   15431.19
6960          1.41    2865414.36   613834.15  1140930.95   23579.86
4636          1.54    1058938.09   469957.60   297119.25    5263.65
2445          1.12      61555.62    17819.26    16576.75    2660.37
8280          1.86     293617.05   196122.05    29716.98      28.49
4260          0.90     228086.10   101029.21    36030.50     403.01
6860          1.01     124946.18     2696.41    45861.16     209.51
6613          0.74    1110895.97   697103.02   161909.48   40611.87
2351          0.83    5236047.86  2351606.01  1833545.05  426600.81
```

6727	1.15	141693.33	47783.40	23081.48	148.91
...	...	...	...	...	...
9163	1.76	1634.59	51.75	93.38	0.00
12154	1.72	13904.53	19.88	578.33	0.00
11007	1.74	16496.39	628.45	9578.24	6.63
9838	1.59	8122.45	391.97	5966.03	0.00
16051	2.32	3748.67	317.51	25.80	0.00
16062	1.19	5887.98	221.91	12.10	0.00
15705	1.82	23190.15	5592.53	28.86	0.00
13949	1.00	8346.57	308.39	2675.89	164.39
14111	1.32	13419.01	883.53	7162.51	12.78
15786	1.06	7728.45	301.90	1339.32	0.00
15992	1.29	4745.15	0.00	1200.84	0.00
15991	1.32	4451.69	0.00	1185.02	0.00
12247	1.64	128204.63	20010.88	59840.22	49.16
12431	1.24	14695.59	459.73	8442.84	0.00
16251	2.44	63800.15	5739.89	25934.55	1805.76
10681	1.78	63537.10	3278.92	23694.61	13.97
18039	1.56	33493.93	2598.21	7330.17	8.57
10141	1.80	1591.60	74.67	1040.59	0.00
17746	1.67	21352.34	10539.17	797.06	33.21
15097	1.63	186112.10	30319.27	66541.46	7.86
15386	2.01	16981.94	4980.49	572.15	26.58
10657	2.07	47878.48	3613.49	20743.86	93.68
16656	1.86	18769.43	1791.82	2951.24	0.00
18157	1.46	115509.80	2796.03	20114.88	105.74
9746	1.34	11247.71	7244.18	1390.20	0.00
17674	1.75	202790.74	29398.11	70514.05	8.08
11415	1.77	3240.71	572.06	1280.97	344.25
17861	1.78	8672.12	1005.73	2110.22	0.00
14284	1.09	150647.80	54851.54	4379.24	0.00
11731	1.78	1983.73	1113.29	63.41	0.00

	Total Bags	Small Bags	Large Bags	XLarge Bags	type	year	\
4006	59340.39	11915.91	47411.82	12.66	conventional	2016	
2449	14000.11	14000.11	0.00	0.00	conventional	2015	
889	31693.36	30800.02	893.34	0.00	conventional	2015	
797	612822.05	457558.59	111788.68	43474.78	conventional	2015	
992	29256.30	17455.74	9975.21	1825.35	conventional	2015	
7986	117600.61	82167.37	31317.19	4116.05	conventional	2017	
3524	113565.65	58374.48	35829.04	19362.13	conventional	2016	
7190	1229648.40	1019859.84	200044.12	9744.44	conventional	2017	
5516	1636666.88	1068035.07	565542.44	3089.37	conventional	2016	
6173	115731.50	98172.10	17016.19	543.21	conventional	2017	
6968	1127351.85	903265.11	187611.42	36475.32	conventional	2017	
2376	638184.59	538970.63	99203.37	10.59	conventional	2015	
7007	59782.33	52292.12	7431.64	58.57	conventional	2017	
589	110266.58	101093.60	9171.11	1.87	conventional	2015	

2632	74648.61	58505.85	16142.76	0.00	conventional	2015
1186	328752.16	311497.60	15650.13	1604.43	conventional	2015
7391	472925.89	276703.90	195838.66	383.33	conventional	2017
1919	59749.80	48200.04	11547.81	1.95	conventional	2015
4530	269424.47	106369.01	163055.46	0.00	conventional	2016
2728	1019569.43	731843.74	286270.91	1454.78	conventional	2015
2247	236162.14	234219.02	0.00	1943.12	conventional	2015
6960	1087069.40	904139.21	145663.00	37267.19	conventional	2017
4636	286597.59	221347.82	56090.36	9159.41	conventional	2016
2445	24499.24	24499.24	0.00	0.00	conventional	2015
8280	67749.53	32356.64	35372.89	20.00	conventional	2017
4260	90623.38	77619.11	13004.27	0.00	conventional	2016
6860	76179.10	22207.37	53245.65	726.08	conventional	2017
6613	211271.60	209810.16	1458.11	3.33	conventional	2017
2351	624295.99	504922.39	119235.17	138.43	conventional	2015
6727	70679.54	29267.91	41411.63	0.00	conventional	2017
...	...	...	...	...	...	...
9163	1489.46	1489.46	0.00	0.00	organic	2015
12154	13306.32	13296.30	10.02	0.00	organic	2016
11007	6283.07	36.85	6246.22	0.00	organic	2015
9838	1764.45	1058.60	705.85	0.00	organic	2015
16051	3405.36	1365.56	2039.80	0.00	organic	2017
16062	5653.97	3727.78	1926.19	0.00	organic	2017
15705	17568.76	17568.76	0.00	0.00	organic	2017
13949	5197.90	1653.33	3544.57	0.00	organic	2016
14111	5360.19	311.44	5048.75	0.00	organic	2016
15786	6087.23	1287.81	4799.42	0.00	organic	2017
15992	3544.31	577.77	2966.54	0.00	organic	2017
15991	3266.67	763.33	2503.34	0.00	organic	2017
12247	48304.37	37233.95	11070.42	0.00	organic	2016
12431	5793.02	923.63	4869.39	0.00	organic	2016
16251	30319.95	30319.95	0.00	0.00	organic	2017
10681	36549.60	36537.93	11.67	0.00	organic	2015
18039	23556.98	1415.85	22087.81	53.32	organic	2018
10141	476.34	326.87	149.47	0.00	organic	2015
17746	9982.90	9087.45	895.45	0.00	organic	2018
15097	89243.51	82024.66	7218.85	0.00	organic	2017
15386	11402.72	11256.31	146.41	0.00	organic	2017
10657	23427.45	23396.97	30.48	0.00	organic	2015
16656	14026.37	1655.19	12367.46	3.72	organic	2017
18157	92493.15	85039.04	7454.11	0.00	organic	2018
9746	2613.33	2613.33	0.00	0.00	organic	2015
17674	102870.50	102717.50	153.00	0.00	organic	2018
11415	1043.43	1033.02	10.41	0.00	organic	2015
17861	5556.17	5538.46	17.71	0.00	organic	2018
14284	91417.02	66104.40	25312.62	0.00	organic	2016
11731	807.03	797.03	10.00	0.00	organic	2015

```
                region
4006         Louisville
2449            Spokane
889  HartfordSpringfield
797          GreatLakes
992         Indianapolis
7986       SouthCarolina
3524            Detroit
7190          Northeast
5516               West
6173          Columbus
6968          Midsouth
2376       SouthCentral
7007          Nashville
589       DallasFtWorth
2632              Tampa
1186         LosAngeles
7391       PhoenixTucson
1919           Portland
4530       PhoenixTucson
2728               West
2247            Seattle
6960          Midsouth
4636            Plains
2445            Spokane
8280              Tampa
4260    NewOrleansMobile
6860          Louisville
6613             Houston
2351       SouthCentral
6727       Jacksonville
...                 ...
9163             Albany
12154            Boston
11007          Portland
9838            Detroit
16051  MiamiFtLauderdale
16062  MiamiFtLauderdale
15705            Houston
13949    RichmondNorfolk
14111           SanDiego
15786       Indianapolis
15992         Louisville
15991         Louisville
12247         California
12431   CincinnatiDayton
16251            NewYork
10681          Northeast
```

```
18039            Portland
10141        Indianapolis
17746              Denver
15097          California
15386              Denver
10657           Northeast
16656            Portland
18157           Southeast
9746        DallasFtWorth
17674          California
11415       SouthCarolina
17861             LasVegas
14284        SouthCentral
11731               Tampa

[100 rows x 12 columns]
```

2.1 Etudes préliminaires des données

- Chargez le jeu de données `orange.csv` dans un dataframe `data`.
- Sauvegardez dans `m` et `d` le nombre d'échantillons et le nombre de variables.

Source [modifiée] : https://husson.github.io/data.html

[77]:
```
orange = pd.read_csv('orange.csv',sep=';',index_col=0)
orange
```

[77]:
```
                Intensite odeur  Typicite odeur  Caractere pulpeux  \
Produit
Pampryl amb.               2.82            2.53               1.66
Tropicana amb.             2.76            2.82               1.91
Fruvita fr.                2.83            2.88               4.00
Joker amb.                 2.76            2.59               1.66
Tropicana fr.              3.20            3.02               3.69
Pampryl fr.                3.07            2.73               3.34

                Intensite gout  Caractere acide  Caractere amer  \
Produit
Pampryl amb.              3.46             3.15            2.97
Tropicana amb.            3.23             2.55            2.08
Fruvita fr.               3.45             2.42            1.76
Joker amb.                3.37             3.05            2.56
Tropicana fr.             3.12             2.33            1.97
Pampryl fr.               3.54             3.31            2.63

                Caractere sucre  Glucose  Fructose  Saccharose  \
Produit
Pampryl amb.               2.60    25.32     27.36       36.45
Tropicana amb.             3.32    17.33     20.00       44.15
```

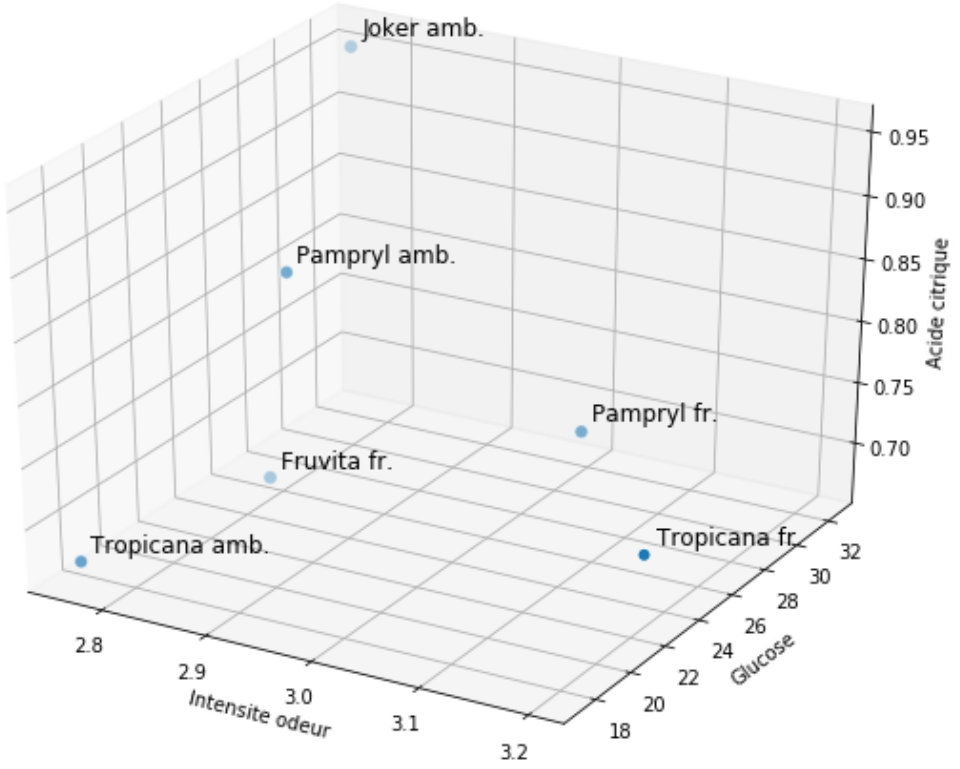
```
Fruvita fr.                3.38    23.65     25.65       52.12
Joker amb.                 2.80    32.42     34.54       22.92
Tropicana fr.              3.34    22.70     25.32       45.80
Pampryl fr.                2.90    27.16     29.48       38.94

                Pouvoir sucrant    pH  Titre  Acide citrique  Vitamine C
Produit
Pampryl amb.              89.95  3.59  13.98            0.84       43.44
Tropicana amb.            82.55  3.89  11.14            0.67       32.70
Fruvita fr.              102.22  3.85  11.51            0.69       37.00
Joker amb.                90.71  3.60  15.75            0.95       36.60
Tropicana fr.             94.87  3.82  11.80            0.71       39.50
Pampryl fr.               96.51  3.68  12.21            0.74       27.00
```

2.1.1 Données centrées et données centrées-réduites

- Visualiser les variables `Intensite odeur`, `Glucose` et `Acide citrique` pour les 6 échantillons du jeux de données

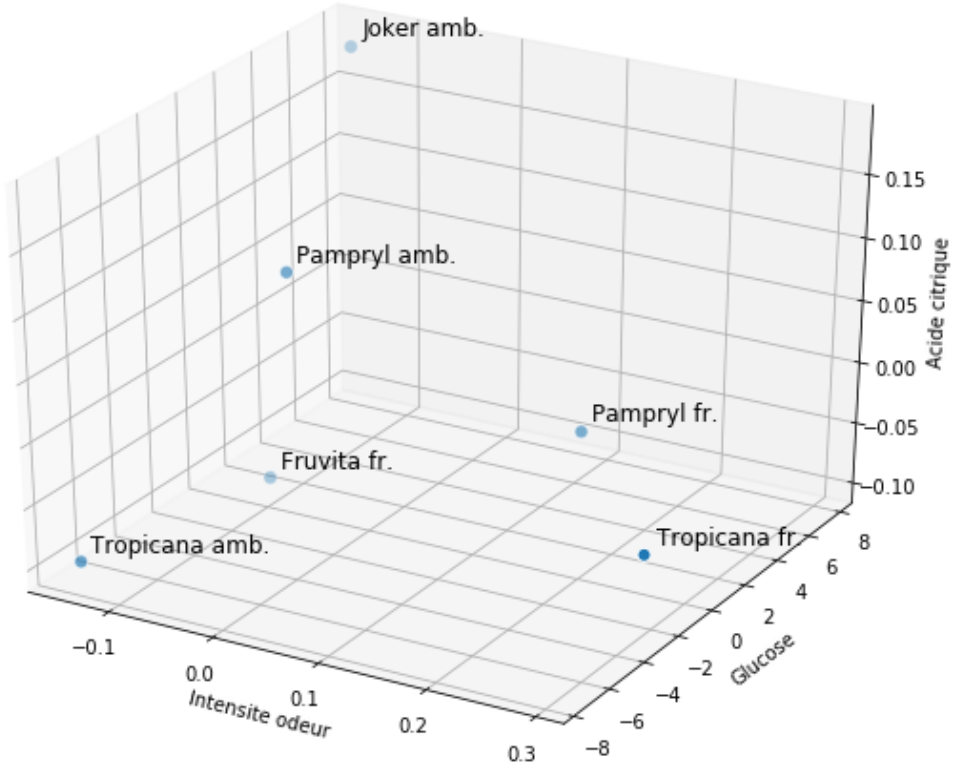
[65]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
```

[85]:
```
def viz3D(data):
    fig = plt.figure(figsize=(8,6))
    ax = Axes3D(fig)

    dx = data['Intensite odeur']
    dy = data['Glucose']
    dz = data['Acide citrique']

    ax.scatter(dx, dy, dz, s=50, edgecolors='w')

    for i, p in enumerate(data.index):
        ax.text(dx[i]+ 0.01, dy[i]+0.01, dz[i]+0.01, p, fontsize=12)

    ax.set_xlabel('Intensite odeur')
    ax.set_ylabel('Glucose')
    ax.set_zlabel('Acide citrique')

    plt.show()
viz3D(orange)
```



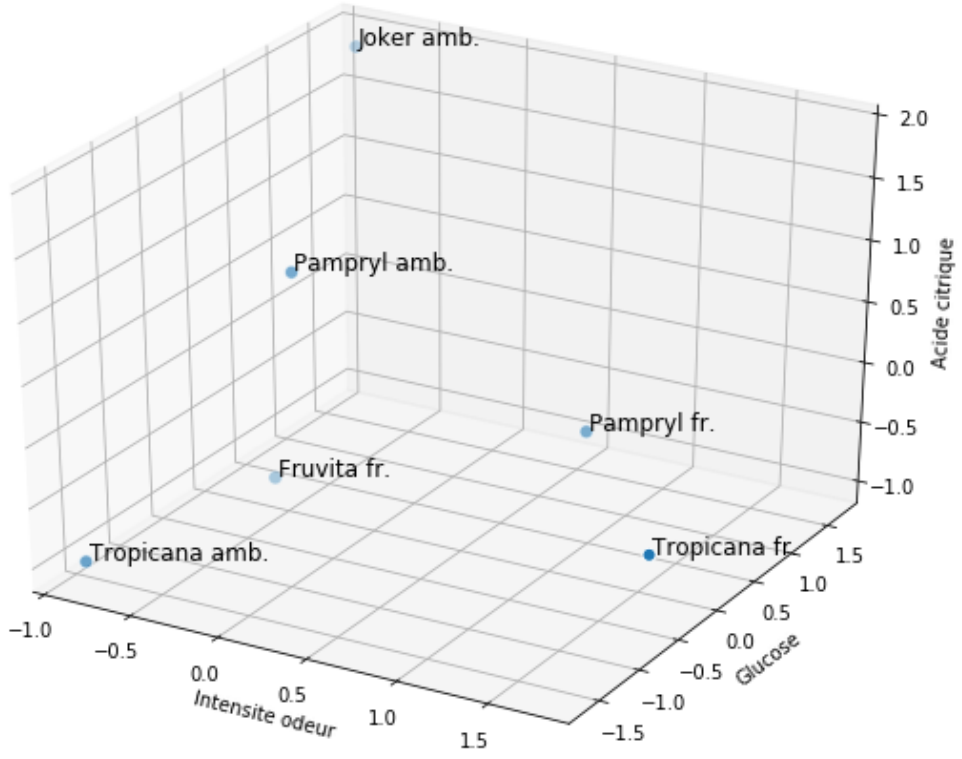
- Centrer l'ensemble des données (soustraire la moyenne) (créer une variable `Y`)
- Revisualiser les données

[86]:
```
Y = orange - orange.mean()
Y
viz3D(Y)
```



- Centrer et réduire l'ensemble des données (soustraire la moyenne et diviser par l'écart-type) (créer une variable `Z`)
- Revisualiser les données

[87]:
```
Z = Y / orange.std(ddof=0)
Z
viz3D(Z)
```



Questions [A COMPLETER] * Quelle est la variance sur chacune des colonnes de Z (données centrées-réduites) ? * Que constatez vous sur les différents axes des trois visualisations ?
* Est ce que la distance entre deux observations est la même pour les données brutes et les données centrées ? et entre les données brutes et les données centrées-réduites ?

[89]:
```
Z.var(ddof=0)
```

[89]:
```
Intensite odeur      1.0
Typicite odeur       1.0
Caractere pulpeux    1.0
Intensite gout       1.0
Caractere acide      1.0
Caractere amer       1.0
Caractere sucre      1.0
Glucose              1.0
Fructose             1.0
Saccharose           1.0
Pouvoir sucrant      1.0
pH                   1.0
Titre                1.0
Acide citrique       1.0
Vitamine C           1.0
```

```
dtype: float64
```

Les données au départ étaient dispersées. Lorsqu'on les centre elles se retrouvent donc autour de 0 en gardant la même importance. Lorsqu'on les réduits, on impose que l'écart entre ces dernières soit le même en gardant les bonnes proportions.

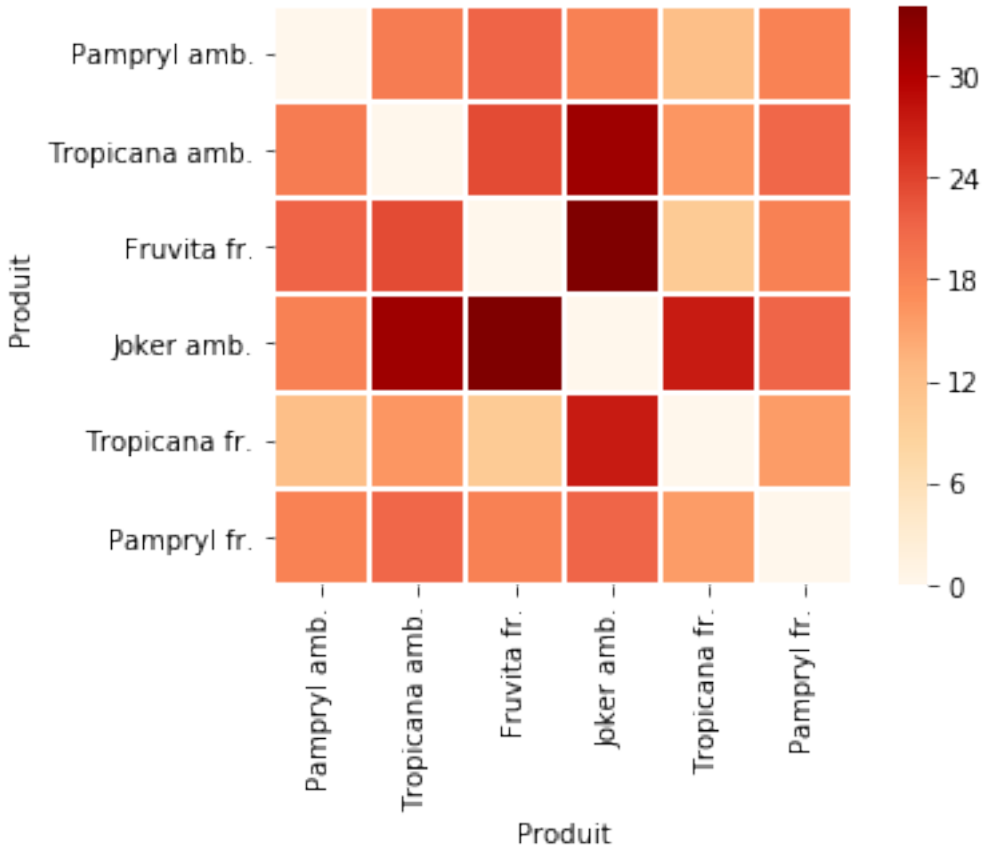
2.1.2 Distance entre les observations

- Calculer la distance entre chacune des observations dans une variable `pairwise`.
- Visualiser cette matrice de distance. On utilisera `heatmap` de la bibliothèque Seaborn.

[97]:
```
from scipy.spatial.distance import pdist
from scipy.spatial.distance import squareform
pairwise = pd.DataFrame(
    squareform(pdist(orange)),
    columns = orange.index,
    index = orange.index
)
```

[100]:
```
sns.heatmap(
    pairwise,
    cmap='OrRd',
    linewidth=1,
    square=True
)
```

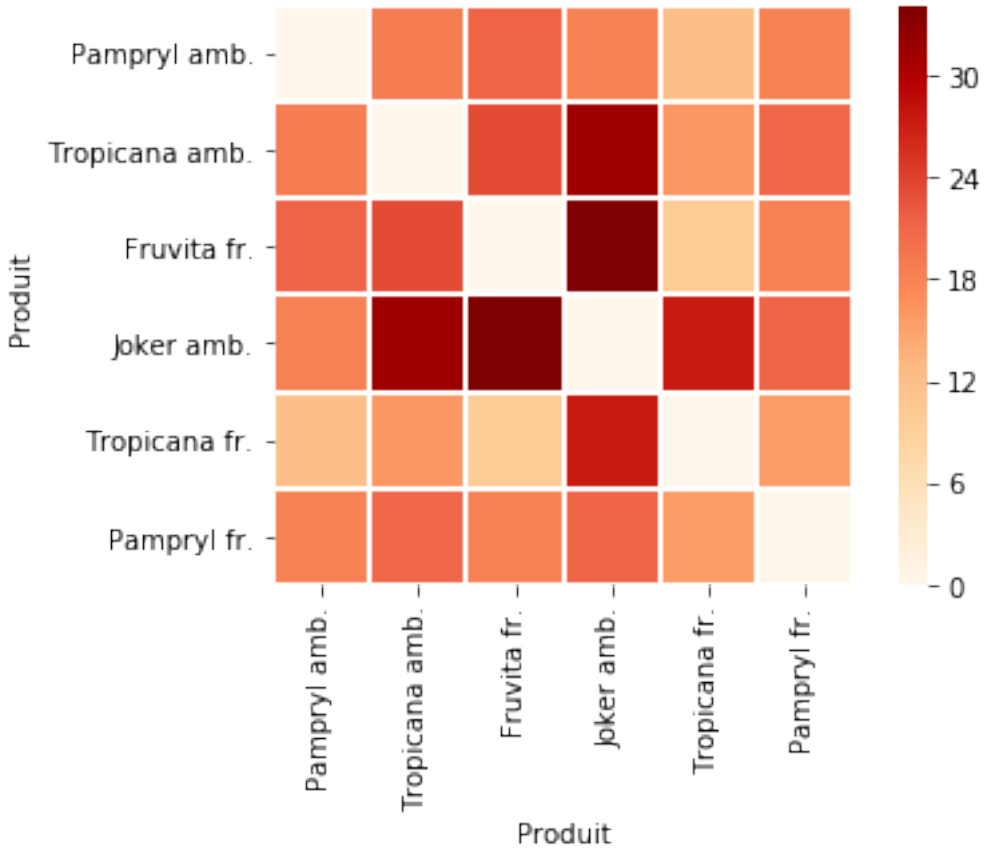
[100]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x244b43fe668>
```



- Répétez ces opérations sur les données centrées (`Y`) et les données centrée-réduites (`Z`).
- Que constatez-vous ?

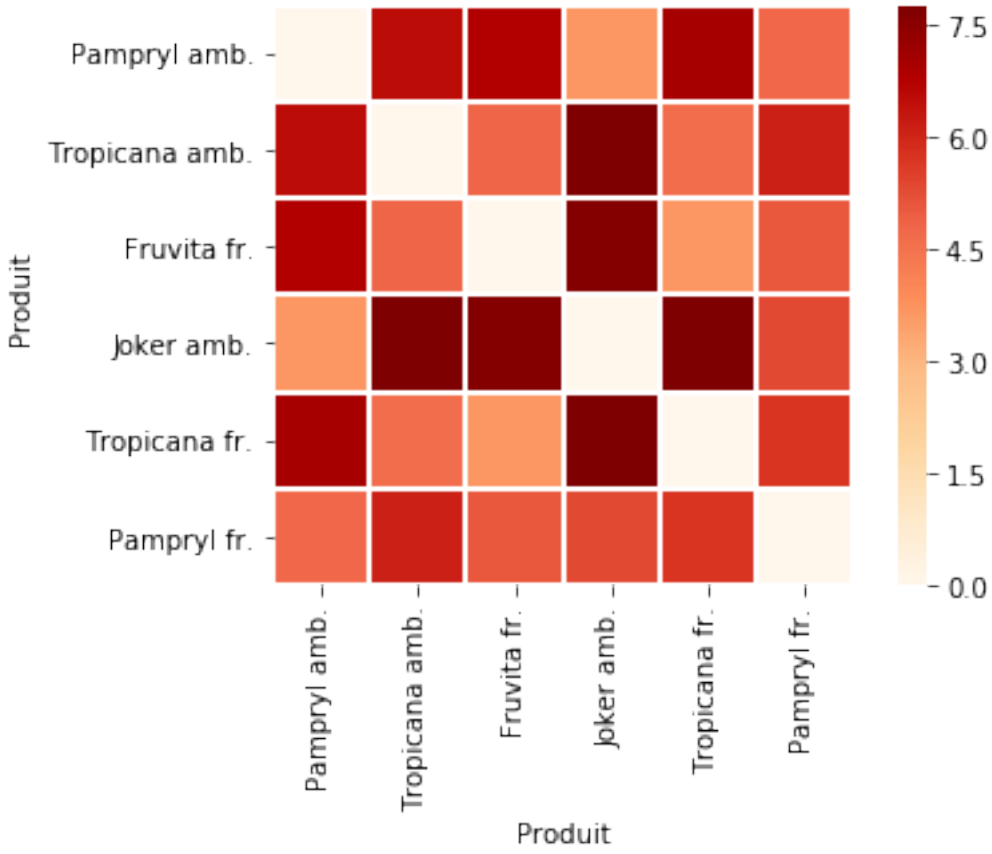
[101]:
```
# Pour Y
pairwiseY = pd.DataFrame(
    squareform(pdist(Y)),
    columns = Y.index,
    index = Y.index
)
sns.heatmap(
    pairwiseY,
    cmap='OrRd',
    linewidth=1,
    square=True
)
```

[101]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x244b453ee10>
```



[102]:
```
# Pour Z
pairwiseZ = pd.DataFrame(
    squareform(pdist(Z)),
    columns = Z.index,
    index = Z.index
)
sns.heatmap(
    pairwiseZ,
    cmap='OrRd',
    linewidth=1,
    square=True
)
```

[102]: `<matplotlib.axes._subplots.AxesSubplot at 0x244b4108f60>`



2.1.3 Corrélation entre les variables

- Utiliser la méthode `corr` de Pandas pour calculer la corrélation entre les différentes variables.
- Quelle est la taille attendue de la matrice de corrélation ?

[ ]:
```
# Calcul des corrélations (dataframe)
```

[ ]:
```
# Taille de la matrice de corrélations
```

[ ]:
```
# Affichage de la matrice de corrélation
```

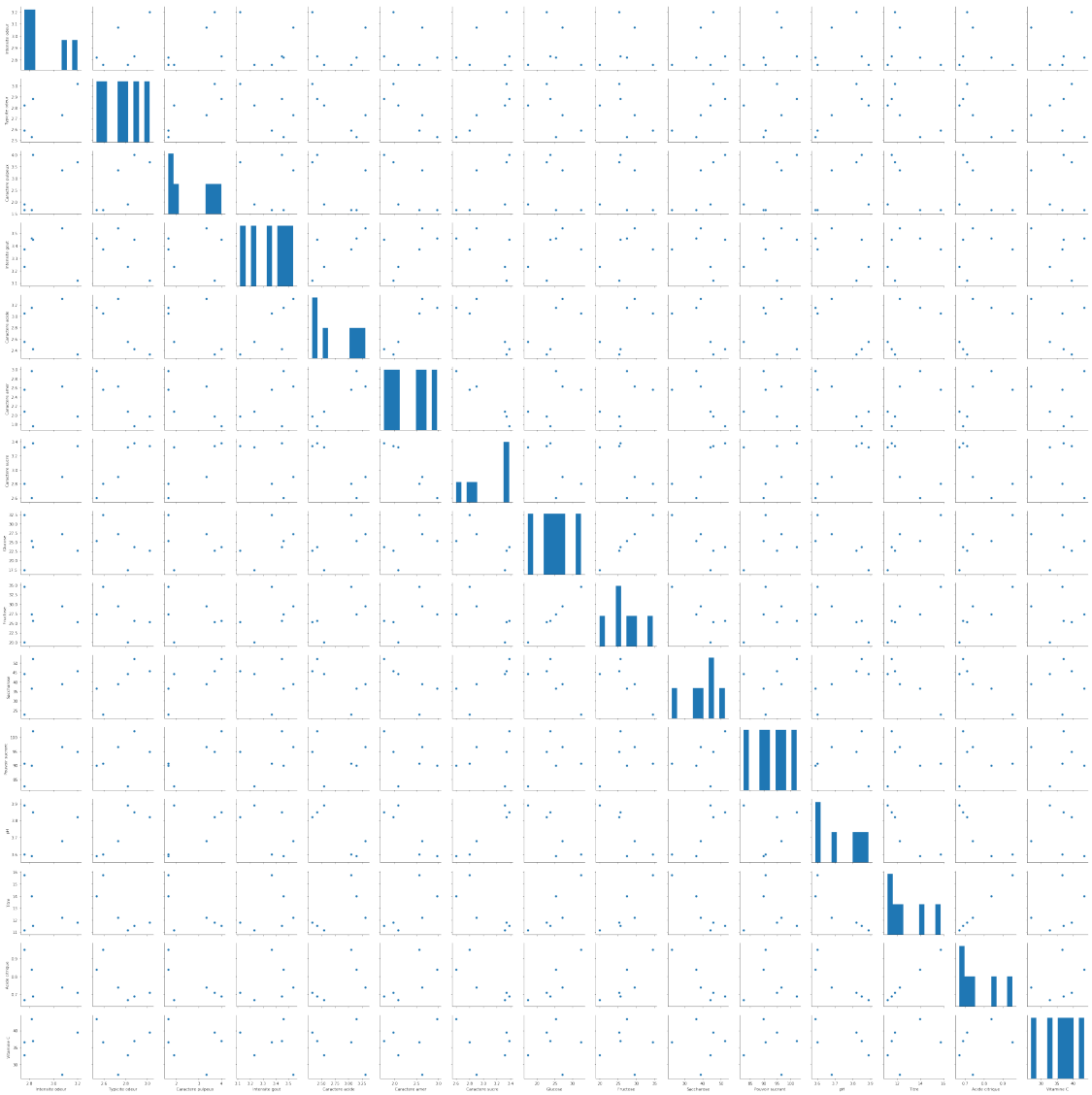
[ ]:
```
# Lien avec la matrice de covariance r_{XY} = \frac{C_{XY}}{\sigma_X\sigma_Y}
```

2.1.4 Visualisation des données

Visualiser les données pour chaque paire d'attributs (i.e., caractéristiques, variables) en utilisant `pairplot` de la bibliothèque Seaborn.

[116]:
```
sns.pairplot(data=orange)
```

[116]:
```
<seaborn.axisgrid.PairGrid at 0x244b3ce7438>
```



Questions [A COMPLETER] On remarque qu'il y a beaucoup de variables (ici `d=15`) pour analyser les données : * Combien de scatterplots indépendants sont ils visibles ? * Combien de scatterplots à analyser si les données étaient dans un espace à `d` dimensions ?

Il y a 225 scatterplots.

2.2 ACP avec Scikit-Learn

Nous allons travailler avec un nouveau jeu de données `decathlon.csv`. Charger les données dans un dataframe `X`.

Créer les dataframes `Y` et `Z` qui correspondent aux données centrées et centrées-réduites, respectivement.

[30]:
```
from pandas import DataFrame, read_csv
import matplotlib.pyplot as plt
```

```
import pandas as pd

data = pd.read_csv('decathlon.csv',sep=';',index_col=0)
data
```

[30]:

	100m	Longueur	Poids	Hauteur	400m	110m H	Disque	Perche	\
Sebrle	10.85	7.84	16.36	2.12	48.36	14.05	48.72	5.0	
Clay	10.44	7.96	15.23	2.06	49.19	14.13	50.11	4.9	
Karpov	10.50	7.81	15.93	2.09	46.81	13.97	51.65	4.6	
Macey	10.89	7.47	15.73	2.15	48.97	14.56	48.34	4.4	
Warners	10.62	7.74	14.48	1.97	47.97	14.01	43.73	4.9	
Zsivoczky	10.91	7.14	15.31	2.12	49.40	14.95	45.62	4.7	
Hernu	10.97	7.19	14.65	2.03	48.73	14.25	44.72	4.8	
Nool	10.80	7.53	14.26	1.88	48.81	14.80	42.05	5.4	
Bernard	10.69	7.48	14.80	2.12	49.13	14.17	44.75	4.4	
Schwarzl	10.98	7.49	14.01	1.94	49.76	14.25	42.43	5.1	
Pogorelov	10.95	7.31	15.10	2.06	50.79	14.21	44.60	5.0	
Schoenbeck	10.90	7.30	14.77	1.88	50.30	14.34	44.41	5.0	
Barras	11.14	6.99	14.91	1.94	49.41	14.37	44.83	4.6	
Smith	10.85	6.81	15.24	1.91	49.27	14.01	49.02	4.2	
Averyanov	10.55	7.34	14.44	1.94	49.72	14.39	39.88	4.8	
Ojaniemi	10.68	7.50	14.97	1.94	49.12	15.01	40.35	4.6	
Smirnov	10.89	7.07	13.88	1.94	49.11	14.77	42.47	4.7	
Qi	11.06	7.34	13.55	1.97	49.65	14.78	45.13	4.5	
Drews	10.87	7.38	13.07	1.88	48.51	14.01	40.11	5.0	
Parkhomenko	11.14	6.61	15.69	2.03	51.04	14.88	41.90	4.8	
Terek	10.92	6.94	15.15	1.94	49.56	15.12	45.62	5.3	
Gomez	11.08	7.26	14.57	1.85	48.61	14.41	40.95	4.4	
Turi	11.08	6.91	13.62	2.03	51.67	14.26	39.83	4.8	
Lorenzo	11.10	7.03	13.22	1.85	49.34	15.38	40.22	4.5	
Karlivans	11.33	7.26	13.30	1.97	50.54	14.98	43.34	4.5	
Korkizoglou	10.86	7.07	14.81	1.94	51.16	14.96	46.07	4.7	
Uldal	11.23	6.99	13.53	1.85	50.95	15.09	43.01	4.5	
Casarsa	11.36	6.68	14.92	1.94	53.20	15.39	48.66	4.4	

	Javelot	1500m
Sebrle	70.52	280.01
Clay	69.71	282.00
Karpov	55.54	278.11
Macey	58.46	265.42
Warners	55.39	278.05
Zsivoczky	63.45	269.54
Hernu	57.76	264.35
Nool	61.33	276.33
Bernard	55.27	276.31
Schwarzl	56.32	273.56
Pogorelov	53.45	287.63

```
Schoenbeck     60.89  278.82
Barras         64.55  267.09
Smith          61.52  272.74
Averyanov      54.51  271.02
Ojaniemi       59.26  275.71
Smirnov        60.88  263.31
Qi             60.79  272.63
Drews          51.53  274.21
Parkhomenko    65.82  277.94
Terek          50.62  290.36
Gomez          60.71  269.70
Turi           59.34  290.01
Lorenzo        58.36  263.08
Karlivans      52.92  278.67
Korkizoglou    53.05  317.00
Uldal          60.00  281.70
Casarsa        58.62  296.12
```

- Quelle est le nombre d'observations dans ces données ?
- Quelle est le nombre de variables ? Que représentent elles?

[14]:
```
print(data.size / data.columns.size)
```

```
28.0
```

[13]:
```
print(data.columns.size)
```

```
10
```

2.2.1 ACP sur les données centrées (Y)

[16]:
```
# Importer le paquetage PCA de la bibliothèque decomposition de Scikit-Learn
from sklearn.decomposition import PCA
```

- Lire la documentation de PCA en ligne

Essayer d'effectuer l'ensemble des actions suivantes 1. Visualiser la valeurs des valeurs propres 2. Visualiser la valeur cumulée des valeurs propres 3. Afficher la valeur des vecteurs propres 4. Visualiser la projection des données sur les deux premières dimensiosn.

[61]:
```
pca = PCA()
comp = pca.fit_transform(data)
df = pd.DataFrame(data = comp, columns = data.columns, index=data.index)
df
```

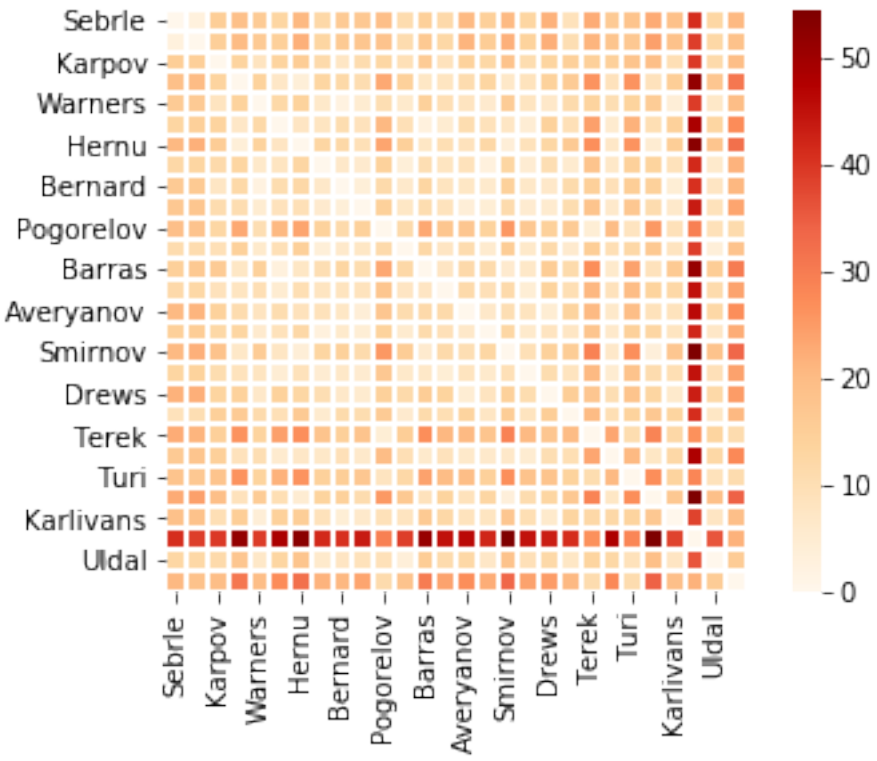
[61]:
```
                100m   Longueur     Poids   Hauteur      400m    110m H  \
Sebrle      1.141275  12.644477  0.253488 -1.638067  0.197933 -0.068912
```

```
Clay          3.338331  12.470786  1.331293 -0.609501 -0.908391 -0.303212
Karpov        1.285006  -0.489211  8.419215 -1.549467 -0.057051  0.155213
Macey       -11.700263  -0.173242  5.163537  0.869885  0.696617  0.054407
Warners       0.806615  -3.499341  0.880096 -1.705432 -0.251970 -0.180224
Zsivoczky    -8.419139   3.881549  0.446142  0.404258  0.425439  0.405231
Hernu       -12.917214  -2.285147  2.102107  0.250694  0.146762 -0.183147
Nool         -1.707879   1.242485 -2.738256 -1.208171 -0.323661  0.339492
Bernard      -0.759255  -3.388650  1.850980 -0.245283  0.213999 -0.291104
Schwarzl     -3.721833  -3.558822 -0.595691  0.152362 -0.139160 -0.480165
Pogorelov    10.754090  -3.979391  1.019789  0.423915  0.733965 -0.617527
Schoenbeck    1.047451   1.959146 -0.840356  0.318774  0.135846 -0.395249
Barras      -11.037209   4.353667 -0.460164  0.422221  0.119203 -0.093178
Smith        -4.811054   3.576301  3.945698  0.713383 -0.044791 -0.294723
Averyanov    -6.155859  -6.329317 -1.991115 -0.124947  0.745396 -0.358296
Ojaniemi     -2.136795  -1.255704 -3.468326 -1.095813  0.729028  0.511172
Smirnov     -14.469277  -0.342082 -1.098627  0.373376 -0.368855  0.212318
Qi           -5.057178   1.384840  0.359126  0.720589 -1.134084  0.152822
Drews        -2.692145  -8.842660 -1.035538 -1.068273 -0.754552 -0.368953
Parkhomenko  -0.557814   5.667672 -4.769854  0.505144  1.322242  0.105716
Terek        13.815702  -6.004802  2.811107 -0.315959  0.590641  0.631402
Gomez        -8.260264  -0.333012 -2.817112 -0.985875  0.216023  0.144975
Turi         12.085255  -0.060292 -5.901652  0.273095 -0.253313 -0.675338
Lorenzo     -14.500317  -3.546889 -2.418380  0.617143 -0.554482  0.724388
Karlivans     1.866717  -5.964663  0.655108  1.268999 -0.699668  0.116567
Korkizoglou  39.931216  -0.990282 -0.368513 -0.813793 -0.181494  0.381472
Uldal         3.952226   0.813757 -2.391193  1.035331 -0.750656  0.247244
Casarsa      18.879612   3.048826  1.657091  3.011412  0.149032  0.127609

               Disque    Perche   Javelot     1500m
Sebrle      -0.112860 -0.024325  0.270019  0.019894
Clay        -0.316985  0.270097 -0.211757 -0.001457
Karpov       0.172551  0.002797 -0.002908  0.009752
Macey       -0.053168  0.274795  0.096976  0.037200
Warners      0.031189  0.051287  0.009769 -0.012614
Zsivoczky   -0.115572 -0.003063 -0.058023  0.121048
Hernu       -0.059075 -0.233808  0.032196  0.044505
Nool        -0.527253 -0.149697 -0.030945 -0.037420
Bernard      0.269015  0.302980 -0.041861  0.088122
Schwarzl    -0.384374 -0.019373  0.106761 -0.045546
Pogorelov   -0.245957  0.039391  0.103362  0.003653
Schoenbeck  -0.261351 -0.050699 -0.033304 -0.123879
Barras       0.192807 -0.269401  0.081750 -0.034643
Smith        0.603791 -0.301456 -0.218121 -0.078424
Averyanov   -0.099973  0.291934 -0.259809 -0.048228
Ojaniemi     0.092978  0.458097 -0.042871 -0.046644
Smirnov     -0.030025 -0.115792 -0.177351  0.040318
Qi           0.012021  0.086534  0.027835  0.041548
```

```
Drews        0.037499 -0.236603  0.012225  0.006391
Parkhomenko  0.102480 -0.228238 -0.010551  0.056563
Terek       -0.386438 -0.437527 -0.065022  0.005306
Gomez        0.519438  0.018954  0.219498 -0.103017
Turi         0.208131 -0.090803 -0.024160  0.132516
Lorenzo      0.038885  0.050156 -0.048992 -0.002599
Karlivans   -0.041740  0.165826  0.258633  0.028902
Korkizoglou  0.310361  0.021940 -0.059899  0.013054
Uldal        0.112952  0.027212  0.040066 -0.042767
Casarsa     -0.069327  0.098784  0.026484 -0.071534
```

[63]:
```
import seaborn as sns
from scipy.spatial.distance import pdist
from scipy.spatial.distance import squareform

pairwise = pd.DataFrame(
    squareform(pdist(df)),
    columns = df.index,
    index = df.index
)
sns.heatmap(
    pairwise,
    cmap='OrRd',
    linewidth=1,
    square=True
)
```

[63]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x15be7eac080>
```

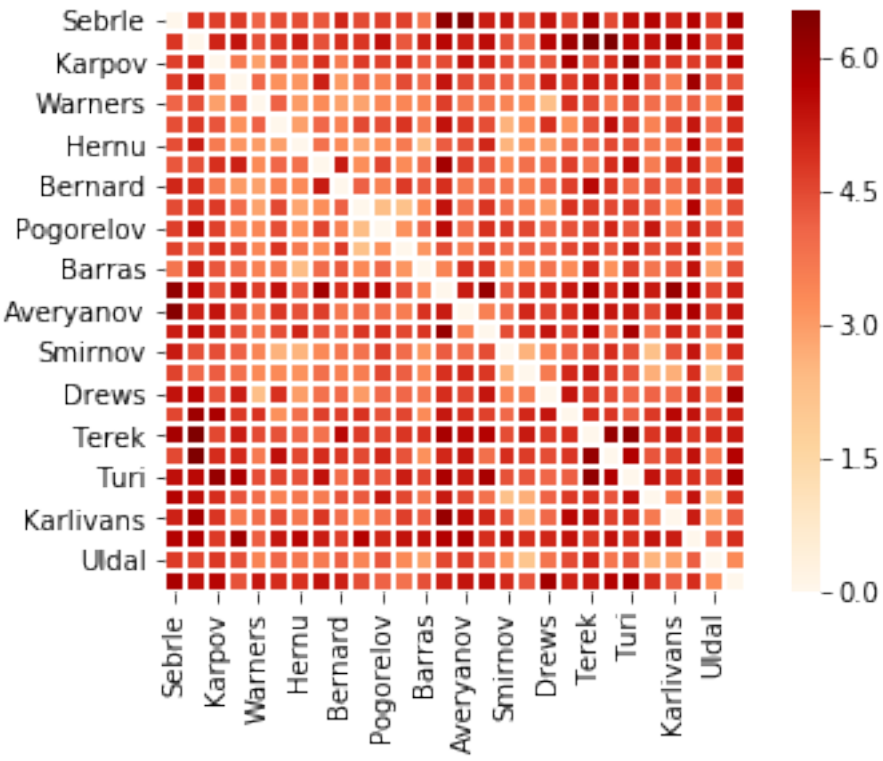


2.3 ACP normée

Recommencer les étapes précédentes sur les données centrées réduites (Z)

[62]:
```
Y = df - df.mean()
Z = Y / df.std(ddof=0)

pairwiseZ = pd.DataFrame(
    squareform(pdist(Z)),
    columns = Z.index,
    index = Z.index
)
sns.heatmap(
    pairwiseZ,
    cmap='OrRd',
    linewidth=1,
    square=True
)
```

[62]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x15be7df64a8>
```



Est ce que les données projetées sont similaires pour l'ACP et l'ACP normée ?

Pour l'ACP de base, les données d'une certaine catégorie contiennent des valeurs qui sont très grandes par rapport aux autres valeurs de cette catégorie. On observe alors de gros écarts dans le heatmap. Lorsqu'on centre l'ACP, les données sont plus proche et on arrive à observer des écarts qui ont du sens.

2.3.1 Complément [pour aller plus loin]

[65]:
```
# Règle du coude et test des batons brisés
##--- Seuils pour le test des bâtons brisés
import numpy as np
bs = 1/np.arange(d,0,-1)
bs = np.cumsum(bs)
bs = bs[::-1]

##--- Test des bâtons brisés
print(pd.DataFrame({'Val.Propre':eigval,'Seuils':bs}))
```

```
␣
↪----------------------------------------------------------------------------
```

```
      NameError                                 Traceback (most recent call␣
↪last)

      <ipython-input-65-4dc554502b40> in <module>
        2 ##--- Seuils pour le test des bâtons brisés
        3 import numpy as np
  ----> 4 bs = 1/np.arange(d,0,-1)
        5 bs = np.cumsum(bs)
        6 bs = bs[::-1]


      NameError: name 'd' is not defined
```

kMeans

May 11, 2020

1 *k*-Moyennes

[2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
```

1.1 Chargement d'un jeux de données

Utilisation des jeux de données disponibles sous Scikit-Learn : https://scikit-learn.org/stable/datasets/index.html

Choisissez un des jeux de données, e.g.

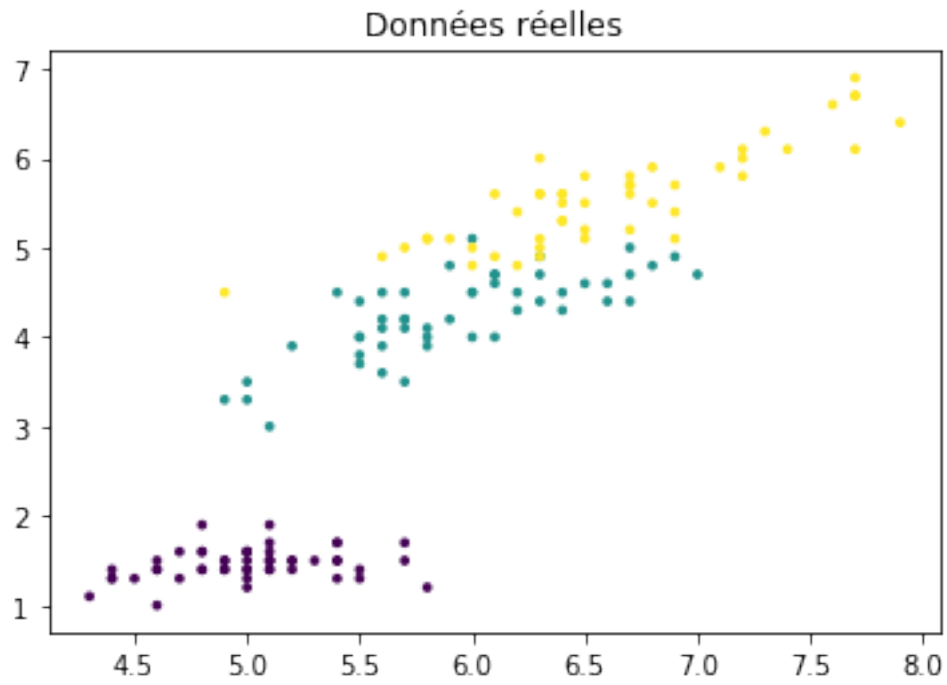
```
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Visualisation
plt.figure()
plt.scatter(X[:,0], y[:,0], s=7, c=y)
plt.show
```

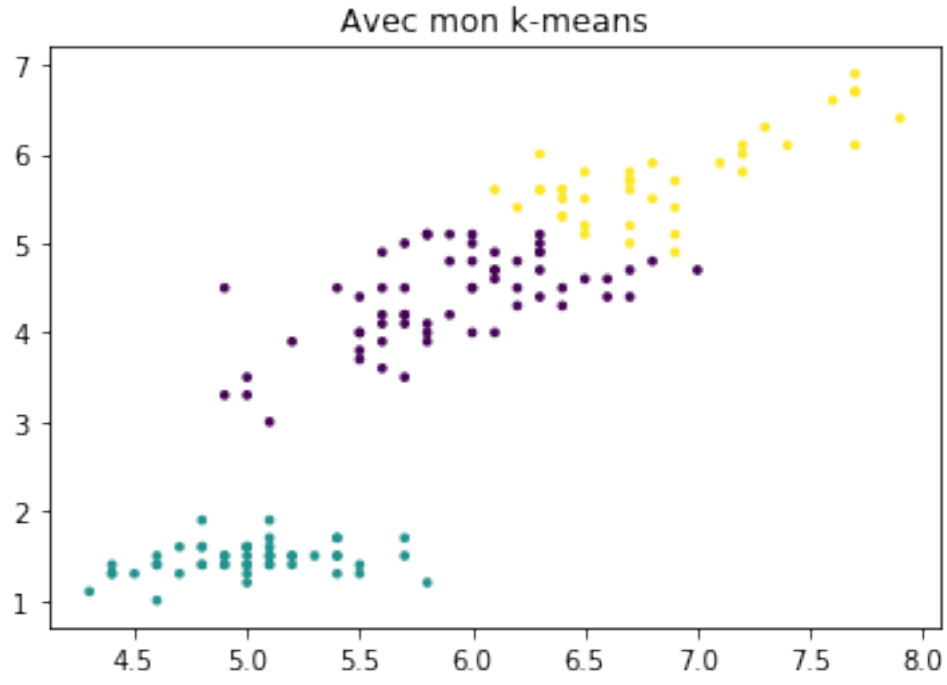
[6]:
```
iris = datasets.load_iris()
X = iris.data
y = iris.target

def init_centroides_random_points(data,nb_clusters):
    max_val = max(np.max(data, axis=0))
    min_val = min(np.min(data, axis=0))
    return np.random.uniform(min_val,max_val,(nb_clusters,data.shape[1]))

def init_centroides_equal_repartition(data,nb_clusters):
    ret = []
    step = (np.floor(len(data) / nb_clusters)).astype(int)
    for i in range(0,len(data),step):
        ret.append(data[i])
```

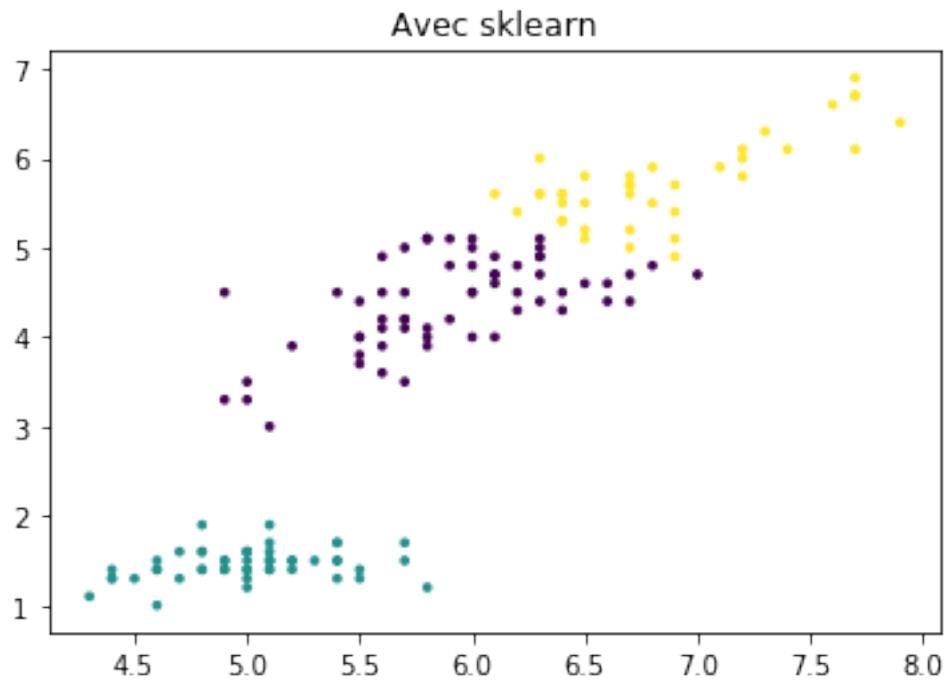
```
    return np.asarray(ret)

def init_centroides_random_index(data,nb_clusters):
    ret = np.zeros((nb_clusters,data.shape[1]))
    for i in range(len(ret)):
        ret[i] = data[np.random.randint(0,data.shape[0]-1)]
    return ret

def affect_clusters(data,centroides,target):
    for i in range(len(data)):
        cluster = 0
        distance = np.linalg.norm(data[i]-centroides[0])
        for j in range(1,len(centroides)):
            new_distance = np.linalg.norm(data[i]-centroides[j])
            if new_distance < distance:
                distance = new_distance
                cluster = j
        target[i] = cluster

def compute_centroides(data,centroides,target):
    for i in range(len(centroides)):
        sum_data = np.zeros(data.shape[1])
        m = 0
        for j in range(len(data)):
            if target[j] == i:
                sum_data += data[j]
                m += 1
        centroides[i] = sum_data / m

def compute_inertia(data,centroides,target):
    ret = 0
    for i in range(len(centroides)):
        for j in range(len(data)):
            if target[j] == i:
                ret += np.sum((data[j] - centroides[i]) ** 2)
    return ret

def k_means(k_clusters_,data_,ITER_MAX_):
    target = np.zeros(len(data_))
    nbIt = 0
    centroides = init_centroides_random_index(data_,k_clusters_)
    new_centroides = []
    while not np.array_equal(centroides,new_centroides) and nbIt < ITER_MAX_:
        affect_clusters(data_,centroides,target)
        new_centroides = centroides.copy()
        compute_centroides(data_,centroides,target)
        nbIt += 1
```

```
    inertia = compute_inertia(data_,centroides,target)
    return target, inertia

def k_means_inertia(k_clusters_,data_,ITER_MAX_, n_repeat):
    target, inertia = k_means(k_clusters_,data_,ITER_MAX_)
    for i in range(1,n_repeat):
        new_target, new_inertia = k_means(k_clusters_,data_,ITER_MAX_)
        if new_inertia < inertia:
            target = new_target.copy()
            inertia = new_inertia
    return target,inertia

def k_means_find_nb_clusters(min_clusters_,max_clusters_,data_,ITER_MAX_,␣
 ↪n_repeat):
    k = min_clusters_
    target,inertia = k_means_inertia(k,data_,ITER_MAX_,n_repeat)
    for i in range(k+1,max_clusters_+1):
        new_target, new_inertia = k_means_inertia(i,data_,ITER_MAX_,n_repeat)
        if new_inertia < inertia:
            k = i
            target = new_target.copy()
            inertia = new_inertia
    return target,k,inertia

target,inertia = k_means_inertia(3,X,200,10)
#target,k,inertia = k_means_find_nb_clusters(3,7,X,200,15)
#print(k)
print(inertia)
skMeans = KMeans(n_clusters=3,random_state=0).fit_predict(X)
# Visualisation
plt.figure(1)
plt.title("Données réelles")
plt.scatter(X[:,0], X[:,2], s=7, c=y)
plt.show()

plt.figure(2)
plt.title("Avec mon k-means")
plt.scatter(X[:,0], X[:,2], s=7, c=target)
plt.show()

plt.figure(3)
plt.title("Avec sklearn")
plt.scatter(X[:,0], X[:,2], s=7, c=skMeans)
plt.show()
```

```
78.85144142614598
```


Données réelles
7
6
5
4
3
2
1
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


Avec mon k-means
7
6
5
4
3
2
1
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


Avec sklearn
7
6
5
4
3
2
1
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

AlgoDescGrad

May 11, 2020

1 TD 4 - La régression linéaire - algo. de descente du gradient

1.1 Packages utiles

[12]:
```
from sklearn import datasets # donnees
import os # rep de travail
import pandas as pd # data analysis
from scipy import stats # stat desc
import matplotlib.pyplot as plt # graphiques
import numpy as np # maths
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
```

1.2 Les données

[5]:
```
#66 Import des données
boston = datasets.load_boston()
print(boston.DESCR)
#0- CRIM      per capita crime rate by town
#1- ZN        proportion of residential land zoned for lots over 25,000 sq.ft.
#2- INDUS     proportion of non-retail business acres per town
#3- CHAS      Charles River dummy variable (= 1 if tract bounds river; 0␣
 ↪otherwise)
#4- NOX       nitric oxides concentration (parts per 10 million)
#5- RM        average number of rooms per dwelling
#6- AGE       proportion of owner-occupied units built prior to 1940
#7- DIS       weighted distances to five Boston employment centres
#8- RAD       index of accessibility to radial highways
#9- TAX       full-value property-tax rate per $10,000
#10- PTRATIO  pupil-teacher ratio by town
#11- B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
#12- LSTAT    % lower status of the population
#13- MEDV     Median value of owner-occupied homes in $1000's
```

```
.. _boston_dataset:
```

Boston house prices dataset

Data Set Characteristics:

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM   per capita crime rate by town
        - ZN   proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS   proportion of non-retail business acres per town
        - CHAS   Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX   nitric oxides concentration (parts per 10 million)
        - RM   average number of rooms per dwelling
        - AGE   proportion of owner-occupied units built prior to 1940
        - DIS   weighted distances to five Boston employment centres
        - RAD   index of accessibility to radial highways
        - TAX   full-value property-tax rate per $10,000
        - PTRATIO pupil-teacher ratio by town
        - B   1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
        - LSTAT   % lower status of the population
        - MEDV   Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/

This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic prices and the demand for clean air', J. Environ. Economics & Management, vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics ...', Wiley, 1980.   N.B. Various transformations are used in the table on pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression

```
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential
Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In
Proceedings on the Tenth International Conference of Machine Learning, 236-243,
University of Massachusetts, Amherst. Morgan Kaufmann.
```

[7]:
```
#-- Récupérer les variables explicatives (X) et leur nom, la variable à␣
 ↪expliquer (Y)
X = boston.data
names = boston.feature_names
Y = boston.target
```

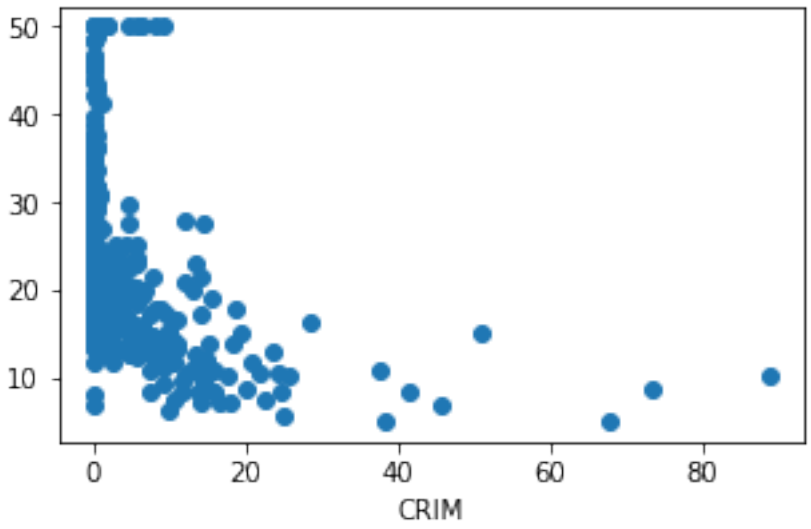
1.2.1 Analyser et sélectionner les données étudiées

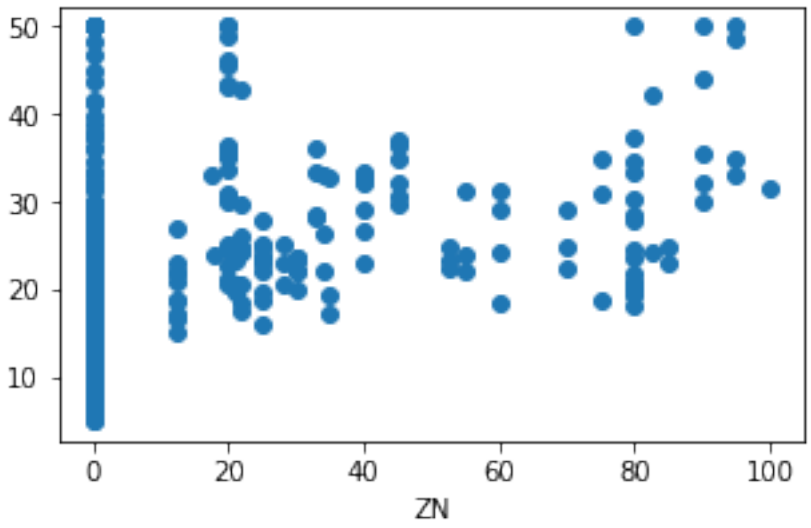
[8]:
```
#-- Afficher les statistiques descriptives
print(stats.describe(X))
print(names)
print(stats.describe(Y))
```

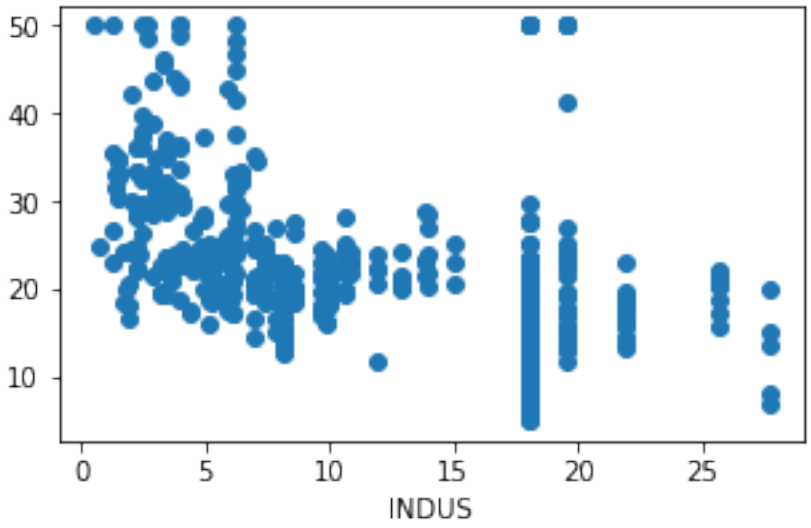
```
DescribeResult(nobs=506, minmax=(array([6.3200e-03, 0.0000e+00, 4.6000e-01,
0.0000e+00, 3.8500e-01,
       3.5610e+00, 2.9000e+00, 1.1296e+00, 1.0000e+00, 1.8700e+02,
       1.2600e+01, 3.2000e-01, 1.7300e+00]), array([ 88.9762, 100.    ,  27.74
,   1.    ,   0.871 ,   8.78  ,
       100.    ,  12.1265,  24.    , 711.    ,  22.    , 396.9   ,
        37.97  ])), mean=array([3.61352356e+00, 1.13636364e+01, 1.11367787e+01,
6.91699605e-02,
       5.54695059e-01, 6.28463439e+00, 6.85749012e+01, 3.79504269e+00,
       9.54940711e+00, 4.08237154e+02, 1.84555336e+01, 3.56674032e+02,
       1.26530632e+01]), variance=array([7.39865782e+01, 5.43936814e+02,
4.70644425e+01, 6.45129730e-02,
       1.34276357e-02, 4.93670850e-01, 7.92358399e+02, 4.43401514e+00,
       7.58163660e+01, 2.84047595e+04, 4.68698912e+00, 8.33475226e+03,
       5.09947595e+01]), skewness=array([ 5.20765239,  2.21906306,  0.29414628,
3.39579929,  0.72714416,
        0.40241467, -0.59718559,  1.00877876,  1.00183349,  0.66796827,
       -0.79994453, -2.88179835,  0.90377074]), kurtosis=array([36.75278626,
3.97994877, -1.23321847,  9.53145284, -0.07586422,
        1.86102697, -0.97001393,  0.47129857, -0.8705205 , -1.14298488,
       -0.29411638,  7.14376929,  0.47654476]))
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
```

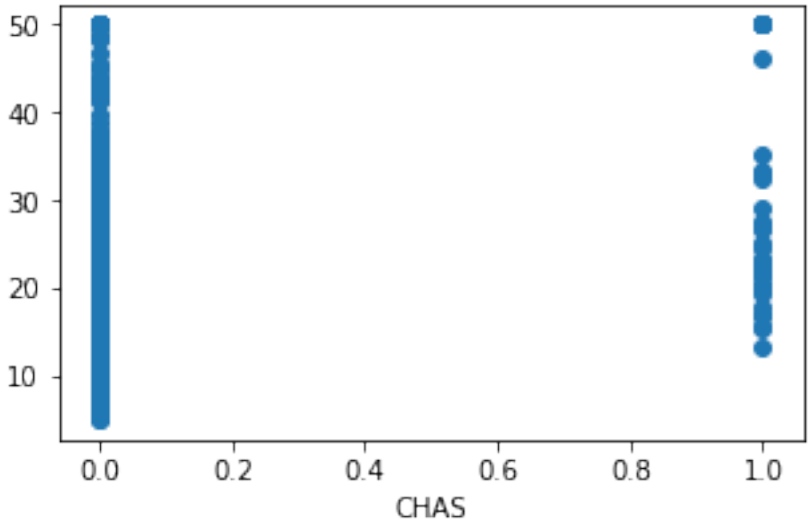
```
DescribeResult(nobs=506, minmax=(5.0, 50.0), mean=22.532806324110677,
variance=84.58672359409856, skewness=1.104810822864635,
kurtosis=1.4686287722747462)
```

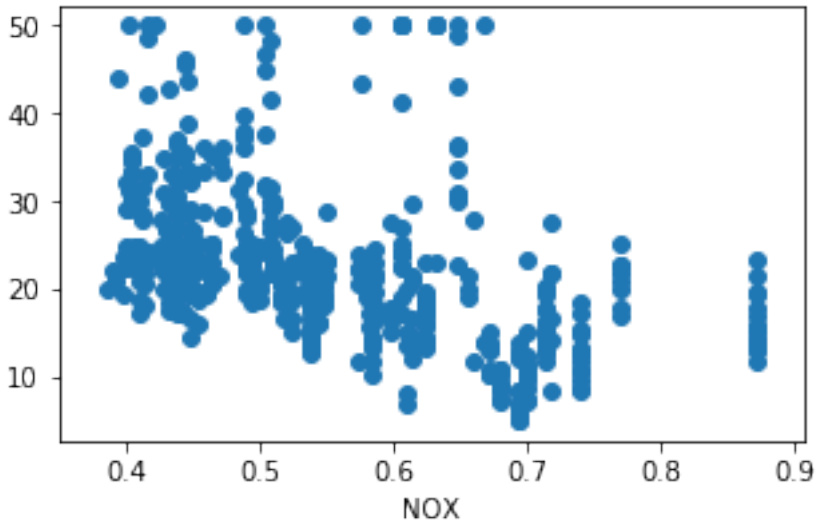
[9]:
```
#-- Sélectionner les variables pour la régression simple puis multiple et
 ↪afficher les (scatterplot)

for i in range(X.shape[1]):
    plt.figure(figsize=(5,3))
    plt.scatter(X[:,i],Y)
    plt.xlabel(names[i])
    plt.show()
```

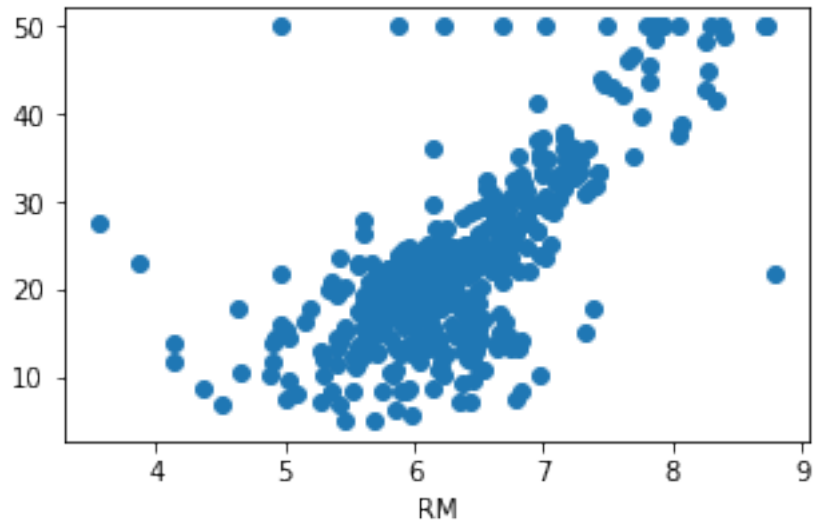


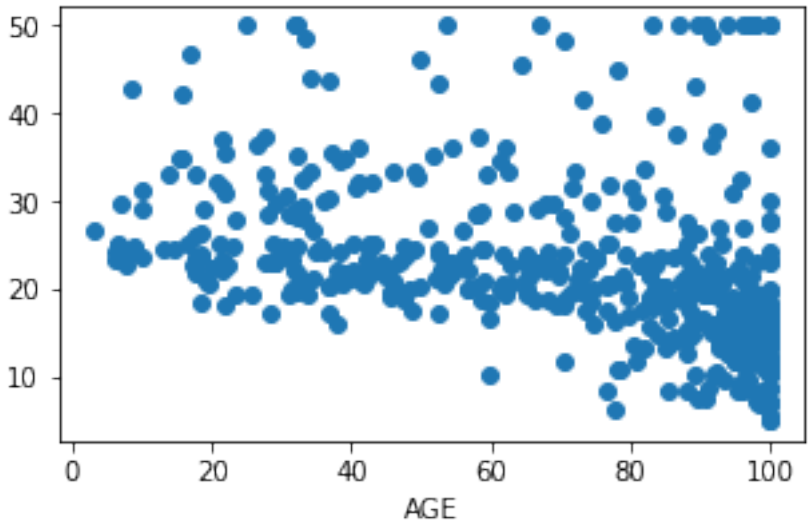


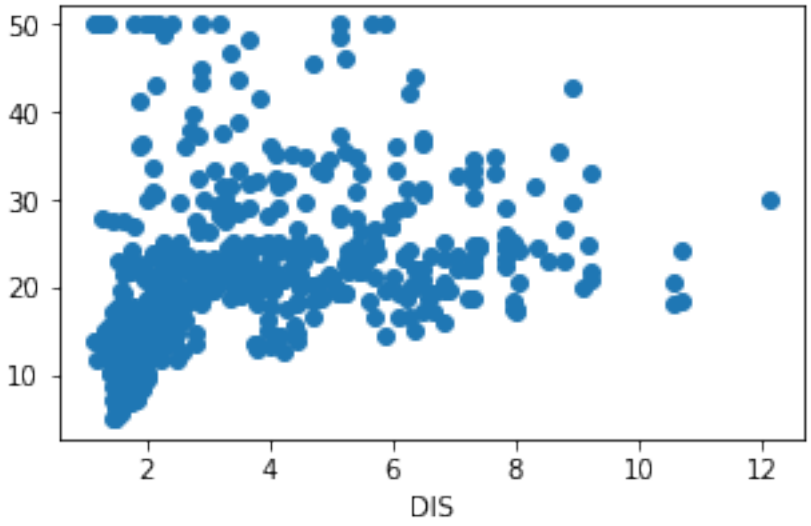

50
40
30
20
10
0
5
10
15
20
25
INDUS

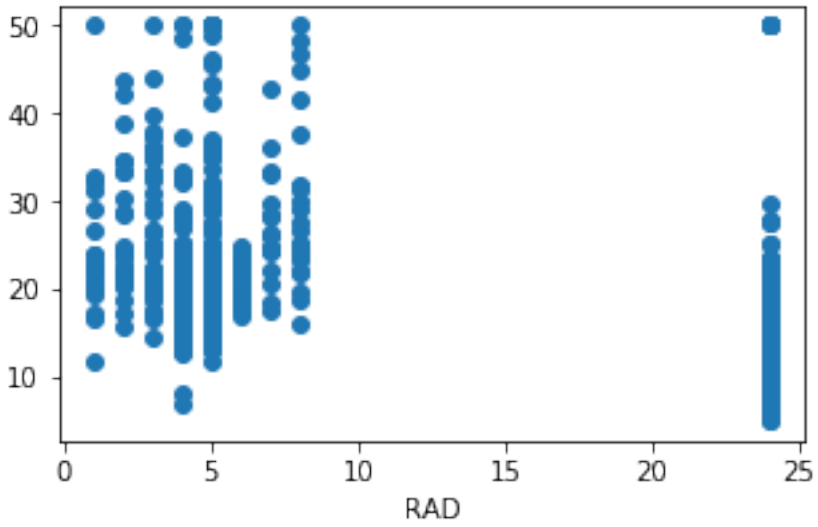

50
40
30
20
10
0.0
0.2
0.4
0.6
0.8
1.0
CHAS

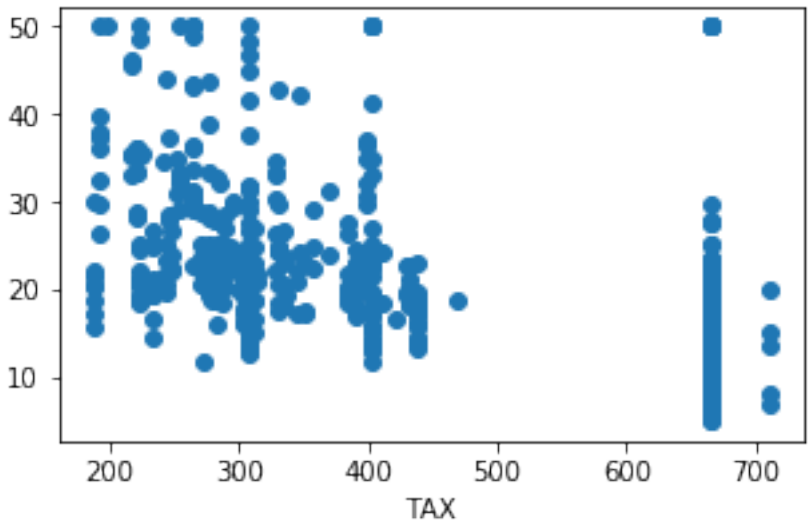

50
40
30
20
10
0.4
0.5
0.6
0.7
0.8
0.9
NOX

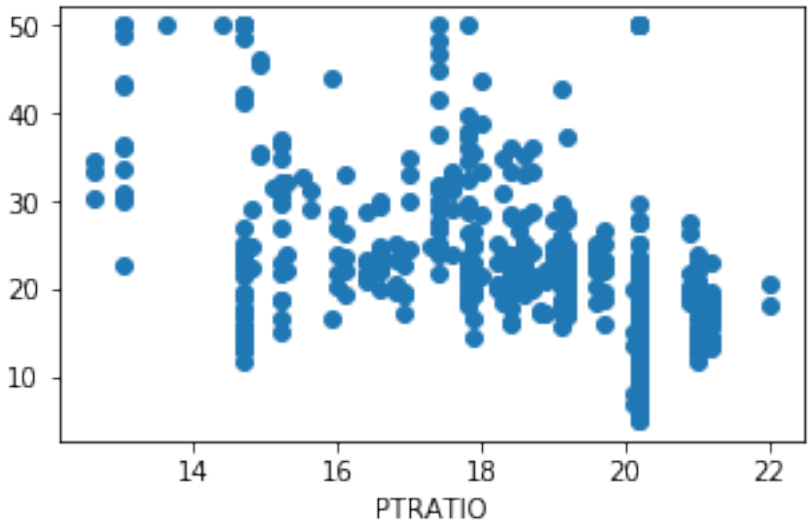

50
40
30
20
10
4
5
6
7
8
9
RM

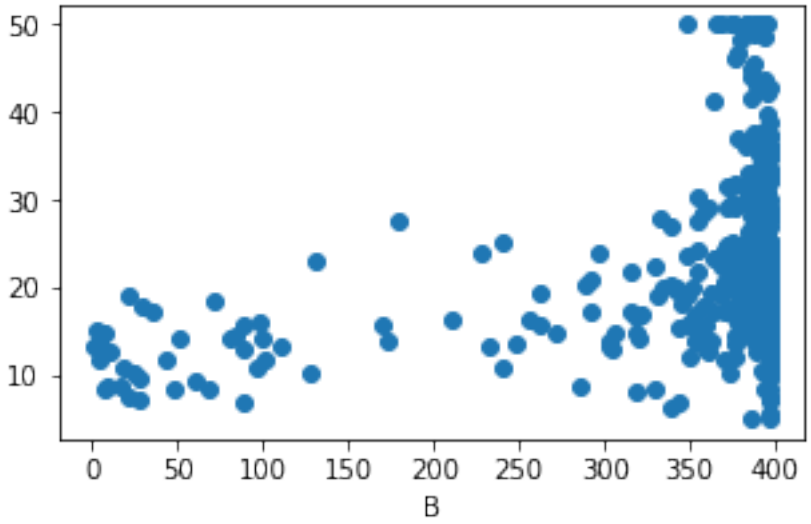

50
40
30
20
10
0
20
40
60
80
100
AGE

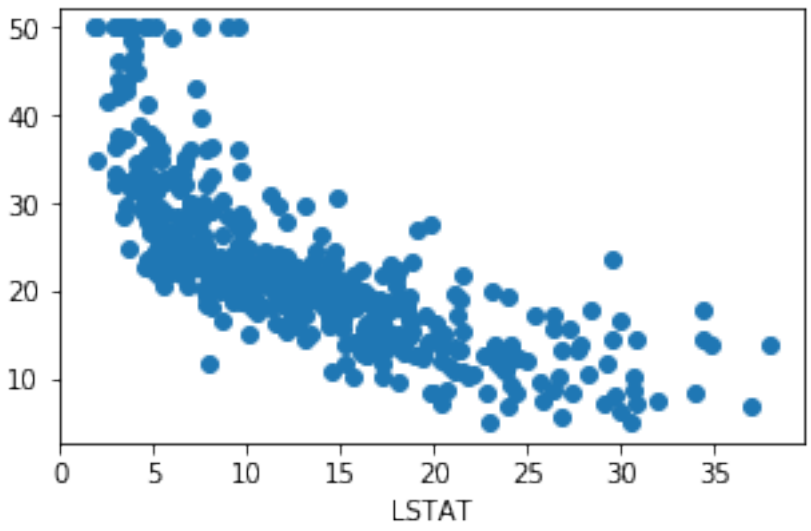

50
40
30
20
10
2
4
6
8
10
12
DIS


50
40
30
20
10
0
5
10
15
20
25
RAD


50
40
30
20
10
200
300
400
500
600
700
TAX


50
40
30
20
10
14
16
18
20
22
PTRATIO


50
40
30
20
10
0
50
100
150
200
250
300
350
400
B



[10]:
```
#-- Préparer les données étudiées
m,d = X.shape
k = 6
kk = 2

xx = np.hstack((np.ones((m,1)),X))
x1 = np.hstack((np.reshape(xx[:,0],(m,1)), np.reshape(xx[:,k],(m,1))))
x2 = np.hstack((np.reshape(xx[:,0],(m,1)), np.reshape(xx[:,kk],(m,1)), np.
 ↪reshape(xx[:,k],(m,1))))
print(x2)
```

```
[[ 1.    18.     6.575]
 [ 1.     0.     6.421]
 [ 1.     0.     7.185]
 ...
 [ 1.     0.     6.976]
 [ 1.     0.     6.794]
 [ 1.     0.     6.03 ]]
```

1.3 Algorithme de descente du gradient

Code des fonctions et tests de ces fonctions dans le cas de la régression simple et multiple
ATTENTION : les paramètres des fonctions sont à compléter

[11]:
```
#-- Modèle

def f(x,B):
    return np.dot(x,B)

#-- Test 1 - reg simple
```

```
B1 = [2,3]
print(f(x1,B1))

#-- Test 2 - reg multiple
B2 = [2,3,1]
print(f(x2,B2))
```

```
[21.725 21.263 23.555 22.994 23.441 21.29  20.036 20.516 18.893 20.012
 21.131 20.027 19.667 19.847 20.288 19.502 19.805 19.97  18.368 19.181
 18.71  19.895 20.426 19.439 19.772 18.797 19.439 20.141 21.485 22.022
 19.139 20.216 19.85  19.103 20.288 19.799 19.523 19.55  19.898 21.785
 23.072 22.31  20.507 20.633 20.207 19.046 19.358 20.09  18.197 18.806
 19.889 20.345 21.533 19.994 19.664 23.747 21.149 22.448 20.435 19.781
 19.223 19.898 21.368 22.286 23.312 20.87  19.361 19.634 18.782 19.655
 21.251 19.883 20.195 20.735 20.819 20.858 20.837 20.42  20.696 19.622
 22.181 21.857 20.906 20.501 21.167 21.89  20.045 20.363 23.021 23.237
 21.251 21.215 21.326 20.633 20.747 21.875 20.489 26.207 25.46  24.248
 22.181 22.343 21.215 20.411 20.501 19.553 19.508 20.381 21.422 20.687
 20.585 22.145 19.739 20.276 20.762 19.784 20.528 20.063 19.616 19.193
 19.61  20.012 19.883 19.568 19.637 19.958 18.839 19.079 21.293 18.911
 21.374 20.978 21.116 19.466 19.271 21.005 19.826 21.362 19.571 20.453
 20.522 17.057 18.209 18.404 16.709 20.39  18.884 16.778 17.558 18.791
 20.366 18.212 17.036 19.127 20.387 20.456 17.816 22.829 20.198 21.53
 20.75  24.467 25.406 27.125 19.562 20.303 25.787 19.631 20.957 21.206
 19.625 19.64  18.716 21.248 19.577 21.638 20.06  20.945 22.58  22.94
 25.295 20.432 23.465 21.689 18.812 20.459 25.493 22.346 21.668 23.555
 22.853 22.217 23.534 22.4   21.812 25.625 23.861 23.321 23.822 22.925
 23.405 20.486 24.83  25.559 26.102 19.673 20.978 19.349 20.192 18.032
 19.88  18.212 19.421 21.125 18.236 20.546 19.664 21.926 19.853 21.119
 22.853 20.492 22.637 21.854 26.798 28.175 26.12  23.489 25.058 21.656
 19.943 24.236 27.011 26.741 22.178 20.258 21.893 24.074 21.443 21.818
 22.691 20.285 21.074 21.179 18.779 18.815 20.324 20.678 21.299 22.154
 21.461 21.314 22.871 26.777 20.324 19.628 24.362 28.112 23.999 22.526
 23.609 24.56  27.194 23.981 23.618 18.68  23.042 26.891 24.41  19.76
 19.568 20.72  21.614 25.073 22.274 22.562 23.801 22.478 21.446 22.436
 25.46  22.904 24.935 25.769 23.264 21.359 20.69  20.627 20.945 21.695
 22.583 23.444 21.89  20.381 20.027 22.034 21.647 19.37  21.035 23.123
 22.613 21.77  21.485 22.946 23.708 21.848 24.26  22.547 21.905 19.916
 16.919 20.366 20.069 20.798 21.701 19.115 19.742 19.346 21.146 20.339
 21.278 21.128 20.123 19.124 21.245 21.293 20.936 20.249 19.604 20.999
 20.432 19.118 20.093 20.948 20.93  20.111 19.607 19.685 20.177 19.955
 19.904 23.723 21.62  22.088 22.622 20.042 19.694 21.548 21.905 22.817
 21.47  21.737 19.652 22.184 18.989 19.808 20.636 21.185 20.381 20.336
 21.194 20.753 18.086 19.409 28.34  12.683 16.889 13.589 16.91  22.049
 23.048 20.648 19.625 16.718 14.414 23.939 21.947 22.382 21.14  20.669
 22.904 21.635 18.608 18.56  15.104 17.831 15.956 17.    16.64  18.17
 19.139 20.153 17.108 20.579 19.661 21.413 21.215 19.241 18.359 19.556
```

```
 19.961 21.029 21.212 18.047 18.593 19.049 14.414 18.824 18.851 22.556
 19.271 21.971 15.884 17.465 15.557 21.302 22.346 17.912 19.871 22.472
 21.233 20.018 18.944 20.309 18.695 19.688 19.511 20.606 20.579 21.14
 21.044 22.499 21.275 21.308 20.624 21.887 21.383 20.456 19.805 18.881
 19.454 21.218 20.657 21.455 19.562 21.377 21.023 20.753 20.555 21.251
 22.247 21.965 20.891 24.179 22.184 21.575 19.928 19.808 20.903 20.243
 22.103 21.128 20.951 21.539 20.627 19.277 19.856 20.009 19.778 19.139
 20.501 20.687 21.311 22.94  18.281 20.486 21.452 17.912 20.555 20.687
 20.726 22.25  23.183 19.286 19.613 20.936 20.342 19.715 18.362 18.242
 17.279 19.949 19.949 19.121 19.778 19.01  18.17  19.382 20.057 18.707
 20.081 21.779 20.36  22.928 22.382 20.09 ]
[ 62.575   8.421   9.185   8.998   9.147   8.43   45.512  45.672  45.131
  45.504  45.877  45.509  45.389   7.949   8.096   7.834   7.935   7.99
   7.456   7.727   7.57    7.965   8.142   7.813   7.924   7.599   7.813
   8.047   8.495   8.674   7.713   8.072   7.95    7.701   8.096   7.933
   7.841   7.85    7.966 233.595 234.024   8.77    8.169   8.211   8.069
   7.682   7.786   8.03    7.399   7.602  70.963  71.115  71.511  70.998
 232.888 279.249 263.383 308.816  83.145  82.927  82.741  82.966  83.456
  83.762  61.604 248.29  247.787  45.378  45.094  45.385   8.417   7.961
   8.065   8.245   8.273   8.286   8.279   8.14    8.232   7.874  83.727
  83.619  83.302  83.167   8.389   8.63    8.015   8.121   9.007   9.079
   8.417   8.405  92.442  92.211  92.249   8.625   8.163  10.069   9.82
   9.416   8.727   8.781   8.405   8.137   8.167   7.851   7.836   8.127
   8.474   8.229   8.195   8.715   7.913   8.092   8.254   7.928   8.176
   8.021   7.872   7.731   7.87    8.004   7.961   7.856   7.879   7.986
   7.613   7.693   8.431   7.637   8.458   8.326   8.372   7.822   7.757
   8.335   7.942   8.454   7.857   8.151   8.174   7.019   7.403   7.468
   6.903   8.13    7.628   6.926   7.186   7.597   8.122   7.404   7.012
   7.709   8.129   8.152   7.272   8.943   8.066   8.51    8.25    9.489
   9.802  10.375   7.854   8.101   9.929   7.877   8.319   8.402   7.875
   7.88    7.572   8.416   7.859   8.546   8.02    8.315   8.86    8.98
   9.765   8.144   9.155   8.563   7.604   8.153   9.831 143.782 143.556
 144.185 143.951 143.739 144.178 188.8   188.604 249.875 249.287 249.107
 249.274 293.975 294.135 255.662 257.11  294.853 295.034   7.891   8.326
   7.783   8.064   7.344   7.96    7.404   7.807   8.375   7.412   8.182
   7.888   8.642   7.951   8.373   8.951   8.164   8.879   8.618  10.266
  10.725  10.04    9.163   9.686   8.552   7.981   9.412  10.337  10.247
   8.726   8.086   8.631   9.358  98.481  98.606  98.897  98.095  98.358
  98.393  73.593  73.605  74.108  74.226  74.433  74.718  74.487  74.438
  74.957  76.259 248.108 247.876 279.454  70.704  69.333  68.842  69.203
  69.52   70.398  69.327  69.206  67.56   69.014  70.297  69.47   67.92
  67.856  68.24   68.538  69.691 128.758 128.854 129.267 128.826 128.482
  68.812  69.82   68.968  69.645 279.923 279.088 173.453 248.23  165.709
 165.815 166.065 248.861 249.148 248.63    8.127   8.009   8.678   8.549
   7.79  218.345 219.041 218.871 110.59  110.495 110.982 108.236 107.616
 108.42  107.849   8.635   7.972   6.973   8.122   8.023   8.266   8.567
   7.705   7.914   7.782   8.382   8.113   8.426   8.376   8.041   7.708
   8.415   8.431   8.312   8.083   7.868   8.333   8.144 112.706 113.031
```

```
  8.316   8.31    8.037   7.869   7.895   8.059   7.985   7.968 114.241
  8.54  173.696 173.874   8.014   7.898 263.516 248.635 128.939 128.49
188.579 187.884 278.728 247.663 247.936   8.212   8.395   8.127   8.112
  8.398   8.251   7.362   7.803  10.78    5.561   6.963   5.863   6.97
  8.683   9.016   8.216   7.875   6.906   6.138   9.313   8.649   8.794
  8.38    8.223   8.968   8.545   7.536   7.52    6.368   7.277   6.652
  7.      6.88    7.39    7.713   8.051   7.036   8.193   7.887   8.471
  8.405   7.747   7.453   7.852   7.987   8.343   8.404   7.349   7.531
  7.683   6.138   7.608   7.617   8.852   7.757   8.657   6.628   7.155
  6.519   8.434   8.782   7.304   7.957   8.824   8.411   8.006   7.648
  8.103   7.565   7.896   7.837   8.202   8.193   8.38    8.348   8.833
  8.425   8.436   8.208   8.629   8.461   8.152   7.935   7.627   7.818
  8.406   8.219   8.485   7.854   8.459   8.341   8.251   8.185   8.417
  8.749   8.655   8.297   9.393   8.728   8.525   7.976   7.936   8.301
  8.081   8.701   8.376   8.317   8.513   8.209   7.759   7.952   8.003
  7.926   7.713   8.167   8.229   8.437   8.98    7.427   8.162   8.484
  7.304   8.185   8.229   8.242   8.75    9.061   7.762   7.871   8.312
  8.114   7.905   7.454   7.414   7.093   7.983   7.983   7.707   7.926
  7.67    7.39    7.794   8.019   7.569   8.027   8.593   8.12    8.976
  8.794   8.03 ]
```

[13]:
```python
#%% Fonction-cout
def cout(x,y,B):
    return (np.sum((f(x,B)-y)**2))/(2*x.shape[0])
#%% Test 1 - reg simple
print(cout(x1,Y,B1))
#%% Test 2 - reg multiple
print(cout(x2,Y,B2))
```

```
32.38256780928853
2461.537040560277
```

[16]:
```python
#%% Gradient
def grad(x,y,B):
    return 1/X.shape[0] * np.dot((f(x,B)-y),x)
#%% Test 1 - reg simple
print(grad(x1,Y,B1))

#%% Test 2 - reg multiple
print(grad(x2,Y,B2))
```

```
[ -1.67890316 -13.55777245]
[  19.84273715 1782.0111996   136.01970681]
```

[17]:
```python
#--- standardisation des X (données centrées-réduites)
# attention : ne pas standardiser X^0...
def standardisation(X):
```

```
    x = X[:,1:]
    return np.hstack((X[:,:1],(x - x.mean(axis=0)) / x.std(axis=0,ddof=0)))
#-- Test 1 - reg simple
print(standardisation(x1))

#-- Test 2 - reg multiple
print(standardisation(x2))
```

```
[[ 1.          0.41367189]
 [ 1.          0.19427445]
 [ 1.          1.28271368]
 ...
 [ 1.          0.98496002]
 [ 1.          0.72567214]
 [ 1.         -0.36276709]]
[[ 1.          0.28482986  0.41367189]
 [ 1.         -0.48772236  0.19427445]
 [ 1.         -0.48772236  1.28271368]
 ...
 [ 1.         -0.48772236  0.98496002]
 [ 1.         -0.48772236  0.72567214]
 [ 1.         -0.48772236 -0.36276709]]
```

[21]:
```
#%% Algo de descente du gradient
def grad_descent(init,x,y,pas,epsi,ITE_MAX):
    nbIt = 0
    x_normalize = standardisation(x)
    cout_arr = []
    b_arr = []
    B = np.asarray(init).astype(float)
    new_B = B.copy()
    b_arr.append(B.tolist())
    old_cout = 0
    new_cout = cout(x_normalize,y,B)
    while(np.abs(new_cout-old_cout)>epsi and nbIt < ITE_MAX):
        new_B = B - pas * grad(x_normalize,y,B)
        B = new_B.copy()
        old_cout = new_cout
        new_cout = cout(x_normalize,y,B)
        b_arr.append(B.tolist())
        cout_arr.append(new_cout)
        nbIt += 1
    return b_arr,cout_arr
#-- Test 1  - reg simple
b, c = grad_descent((0,0),x1,Y,0.01,0.01,500)
print(b)
print(c)
```

```
#-- Test 2 - reg multiple
print()
b, c = grad_descent((0,0,0),x2,Y,0.05,0.01,500)
print(b)
print(c)
```

```
[[0.0, 0.0], [0.22532806324110696, 0.06388975221817343], [0.4484028458498027,
0.12714060691416512], [0.6692468806324112, 0.18975895306319684],
[0.8878824750671941, 0.2517511157507382], [1.1043317135576287,
0.31312335681140424], [1.3186164596631589, 0.3738818754614636],
[1.530758358307634, 0.4340328089250224], [1.7407788379656646,
0.4935822330539456], [1.9486991128271147, 0.5525361629415795],
[2.1545401849399504, 0.6109005535303371], [2.3583228463316575,
0.6686813002132072], [2.5600676811094476, 0.7258842394292485],
[2.75979506753946, 0.7825151492531295], [2.9575251801051716,
0.8385797499787716], [3.1532779915452265, 0.8940837046971573],
[3.3470732748708807, 0.9490326198683591], [3.5389306053632783,
1.003432045887849], [3.728869362550752, 1.057287477647144], [3.916908732166351,
1.1106043550888458], [4.103067708085795, 1.1633880637561307],
[4.287365094246043, 1.2156439353367428], [4.46981950654469, 1.2673772482015486],
[4.65044937472035, 1.3185932279377066], [4.829272944214253, 1.369297047876503],
[5.0063082780132175, 1.4194938296159114], [5.181573258474192,
1.4691886435379258], [5.355085589130557, 1.5183865093207198],
[5.526862796480358, 1.567092396445686], [5.696922231756661, 1.6153112246994026],
[5.865281072680201, 1.663047864670582], [6.031956325194506, 1.7103071382420496],
[6.1969648251836675, 1.7570938190778027], [6.360323240172938,
1.803412633105198], [6.522048071012315, 1.8492682589923195], [6.682155653543298,
1.8946653286205697], [6.840662160248972, 1.9396084275525374],
[6.997583601887589, 1.9841020954951856], [7.15293582910982, 2.028150826758407],
[7.306734534059829, 2.0717590707089966], [7.458995251960338, 2.114931232220008],
[7.609733362681841, 2.1576716721160527], [7.75896409229613, 2.1999847076130656],
[7.906702514614275, 2.2418746127551086], [8.052963552709238, 2.283345618845731],
[8.197761980423254, 2.324401914875447], [8.341112423860128, 2.365047647944866],
[8.483029362862634, 2.4052869236835908], [8.623527132475115,
2.4451238066649283], [8.76261992439147, 2.4845623208164525], [8.900321788388663,
2.5236064498264614], [9.036646633745884, 2.56226013754637], [9.17160823064953,
2.6005272883890798], [9.305220211584142, 2.6384117677233623],
[9.437496072709408, 2.675917402264302], [9.56844917522342, 2.7130479804598324],
[9.698092746712293, 2.7498072528734077], [9.826439882486277, 2.786198932562847],
[9.953503546902521, 2.822226695455392], [10.079296574674602,
2.8578941807190117], [10.203831672168963, 2.893204991129995],
[10.32712141868838, 2.9281626934368683], [10.449178267742603,
2.962770818720673], [10.570014548306284, 2.99703286275164], [10.689642466064328,
3.030952286342297], [10.80807410464479, 3.0645325156970475], [10.92532142683945,
3.0977769427582507], [11.041396275812161, 3.1306889255488417],
[11.156310376295146, 3.1632717885115267], [11.270075335773301,
3.195528822844585], [11.382702645656675, 3.2274632868343125],
```

[11.494203682441215, 3.259078406184143], [11.60458970885791, 3.290377374340475],
[11.713871875010438, 3.3213633528152435], [11.82206121950144,
3.3520394715052646], [11.929168670547533, 3.3824088290083854],
[12.035205047083164, 3.4124744929364748], [12.140181059853438,
3.4422395002252832], [12.244107312496011, 3.471706857441204],
[12.346994302612158, 3.5008795410849656], [12.448852422827143,
3.529760497892289], [12.549691961839978, 3.55835264513154], [12.649523105462684,
3.5866588708983977], [12.748355937649164, 3.614682034407587],
[12.846200441513778, 3.6424249662816846], [12.943066500339746,
3.6698904688370413], [13.038963898577455, 3.697081316366844],
[13.133902322832787, 3.724000255421349], [13.227891362845567,
3.750650005085309], [13.320940512458218, 3.7770332572526293],
[13.413059170574742, 3.8031526768982764], [13.504256642110102,
3.829010902347467], [13.594542138930107, 3.854610545542166],
[13.683924780781913, 3.879954192304918], [13.7724135962152, 3.905044402600042],
[13.860017523494156, 3.929883710792215], [13.94674541150032, 3.954474625902466],
[14.032606020626423, 3.978819631861615], [14.117608023661266,
4.002921187761173], [14.20176000666576, 4.026781728101734], [14.285070469840209,
4.050403663038891], [14.367547828382913, 4.0737893786266755],
[14.44920041334019, 4.096941237058582], [14.530036472447895, 4.119861576906169],
[14.610064170964522, 4.142552713355281], [14.689291592495984,
4.165016938439901], [14.767726739812131, 4.187256521273675],
[14.845377535655116, 4.209273708279112], [14.922251823539671,
4.231070723414494], [14.998357368545381, 4.2526497683985225],
[15.073701858101034, 4.274013022932711], [15.14829290276113, 4.295162644921557],
[15.222138036974625, 4.3161007706905155], [15.295244719845986,
4.336829515201783], [15.367620335888633, 4.357350972267939],
[15.439272195770853, 4.3776672147634335], [15.510207537054251,
4.397780294833972], [15.580433524924816, 4.417692244103806],
[15.649957252916675, 4.437405073880941], [15.718785743628615,
4.456920775360305], [15.786925949433435, 4.476241319824876],
[15.854384753180208, 4.495368658844801], [15.921168968889512,
4.514304724474527], [15.987285342441725, 4.533051429447955],
[16.052740552258413, 4.551610667371649], [16.117541209976935,
4.569984312916105], [16.181693861118273, 4.588174222005118],
[16.245204985748195, 4.60618223200324], [16.30808099913182, 4.624010161901381],
[16.37032825238161, 4.641659812500541], [16.431953033098903, 4.65913296659371],
[16.492961566009022, 4.676431389145946], [16.553360013590037,
4.693556827472659], [16.613154476695243, 4.710511011416107], [16.6723509951694,
4.727295653520119], [16.730955548458812, 4.743912449203091], [16.78897405621533,
4.7603630769292336], [16.846412378894286, 4.776649198378115],
[16.90327631834645, 4.792772458612507], [16.95957161840409, 4.808734486244556],
[17.015303965461158, 4.824536893600284], [17.070478989047654,
4.840181276882454], [17.125102262398283, 4.855669216331803],
[17.179179303015406, 4.871002276386658], [17.232715573226358,
4.886182005840965], [17.2857164807352, 4.901209938000728], [17.338187379168954,
4.916087590838894], [17.39013356861837, 4.930816467148679], [17.441560296173293,
4.9453980546953655], [17.492472756452667, 4.9598338263665855],

[17.542876092129248, 4.974125240321093], [17.592775394449063,
4.988273740136055], [17.64217570374568, 5.002280754952868], [17.69108200994933,
5.016147699621513], [17.739499253090944, 5.029875974843471], [17.78743232380114,
5.043466967313211], [17.834886063804237, 5.056922049858252], [17.881865266407303,
5.070242581577843], [17.928374676984337, 5.083429907980238], [17.9744189934556,
5.096485361118609], [18.02000286676215, 5.109410259725596], [18.065130901335635,
5.122205909346514], [18.109807655563387, 5.134873602471222], [18.15403764224886,
5.147414618664683], [18.197825329067477, 5.15983022469621], [18.24117513901791,
5.172121674667422], [18.284091450868836, 5.184290210138921],
[18.326578599601255, 5.196337060255705], [18.36864087684635, 5.208263441871321],
[18.41028253131899, 5.2200705596707815], [18.45150776924691, 5.231759606292247],
[18.492320754795546, 5.2433317624474975], [18.532725610488697,
5.254788197041196], [18.572726417624917, 5.266130067288958],
[18.612327216689774, 5.277358518834242], [18.651532007763983,
5.288474685864073], [18.69034475092745, 5.299479691223605], [18.728769366659282,
5.310374646529542], [18.766809736233796, 5.32116065228242], [18.804469702112566,
5.3318387979777695], [18.841753068332547, 5.342410162216165],
[18.878663600890327, 5.352875812812177], [18.915205028122532,
5.363236806902228], [18.951381041082414, 5.373494191051379],
[18.987195293912695, 5.383649001359039], [19.022651404214674,
5.393702263563623], [19.057752953413633, 5.4036549931461595],
[19.092503487120602, 5.413508195432871], [19.126906515490504,
5.4232628656967155], [19.160965513576706, 5.432919989257922],
[19.194683921682046, 5.4424805415835165], [19.22806514570633,
5.451945488385855], [19.261112557490375, 5.46131578572017], [19.293829495156576,
5.470592380081142], [19.326219263446117, 5.479776208498504], [19.35828513405276,
5.4888681986316925], [19.39003034595334, 5.497869268863549], [19.42145810573491,
5.506780328393087], [19.45257158791867, 5.515602277327329], [19.48337393528059,
5.52433600677223], [19.51386825916889, 5.532982398922681], [19.54405763981831,
5.541542327151628], [19.573945126661233, 5.550016656098285],
[19.603533738635726, 5.558406241755476], [19.632826464490474,
5.566711931556094], [19.661826263086677, 5.574934564458706],
[19.690536063696918, 5.583074971032293], [19.718958766301057,
5.591133973540143], [19.747097241879153, 5.599112386022915], [19.77495433270147,
5.607011014380859], [19.80253285261556, 5.614830656455224], [19.82983558733051,
5.622572102108845], [19.856865294698313, 5.63023613330593], [19.883624704992435,
5.637823524191044], [19.910116521183618, 5.645335041167307], [19.93634341921289,
5.652771442973807], [19.962308048261868, 5.6601334807622425],
[19.988013031020355, 5.667421898172794], [20.013460963951257, 5.67463743140924],
[20.038654417552852, 5.68178080931332], [20.06359593661843, 5.68885275343836],
[20.088288040493353, 5.69585397812215], [20.112733223329528, 5.702785190559102],
[20.13693395433734, 5.709647090871685], [20.160892678035076, 5.716440372181141],
[20.18461181449583, 5.723165720677503], [20.20809375959198, 5.729823815688902],
[20.231340885237167, 5.736415329750186], [20.2543555396259, 5.7429409286708575],
[20.27714004747075, 5.749401271602323], [20.299696710237146, 5.755797011104473],
[20.322027806375882, 5.762128793211602], [20.34413559155323, 5.768397257497659],
[20.366022298878804, 5.774603037140856], [20.387690139131124,
5.780746758987621], [20.40914130098092, 5.786829043615918], [20.43037795121222,

5.792850505397932], [20.451402234941202, 5.798811752562126],
[20.472216275832896, 5.804713387254679], [20.492822176315673,
5.810556005600305], [20.513222017793623, 5.816340197762475],
[20.533417860856794, 5.822066548003024], [20.55341174548933, 5.827735634741167],
[20.573205691275543, 5.833348030611929], [20.592801697603896,
5.838904302523983], [20.612201743868965, 5.844405011716916],
[20.631407789671382, 5.84985071381792], [20.650421775015776, 5.855241958897914],
[20.669245620506725, 5.860579291527109], [20.687881227542764,
5.865863250830011], [20.706330478508445, 5.871094370539884], [20.72459523696447,
5.876273179052659], [20.74267734783593, 5.881400199480306], [20.760578637598677,
5.8864759497036765], [20.778300914463795, 5.8915009424248135],
[20.795845968560265, 5.896475685218739], [20.81321557211577, 5.901400680584725],
[20.830411479635718, 5.906276425997051], [20.847435428080466,
5.911103413955254], [20.864289137040767, 5.915882132033874],
[20.880974308911465, 5.920613062931709], [20.897492629063457,
5.925296684520565], [20.913845766013928, 5.929933469893533],
[20.930035371594897, 5.934523887412771], [20.946063081120055,
5.939068400756817], [20.961930513549962, 5.943567468967422], [20.97763927165557,
5.948021546495921], [20.99319094218012, 5.952431083249135], [21.008587095999427,
5.956796524634817], [21.023829288280538, 5.961118311606643], [21.03891905863884,
5.96539688070875], [21.053857931293557, 5.969632664119835], [21.06864741522173,
5.973826089696811], [21.08328900431062, 5.977977581018016], [21.09778417750862,
5.982087557426009], [21.11213439897464, 5.9861564340699225],
[21.126341118225998, 5.990184621947397], [21.140405770284843,
5.9941725279460965], [21.1543297758231, 5.998120554884809], [21.168114541305975,
6.002029101554134], [21.181761459134023, 6.005898562756767], [21.19527190778379,
6.009729329347373], [21.20864725194706, 6.013521788272072], [21.221888842668697,
6.017276322607525], [21.234998017483118, 6.020993311599623],
[21.247976100549394, 6.0246731307018], [21.26082440278501, 6.028316151612956],
[21.273544221998264, 6.031922742315], [21.28613684301939, 6.035493267110024],
[21.298603537830303, 6.039028086657097], [21.310945565693107,
6.0425275580086995], [21.32316417327728, 6.045992034646786],
[21.335260594785616, 6.049421866518491], [21.347236052078866, 6.05281740007148],
[21.359091754799184, 6.056178978288939], [21.3708289004923, 6.059506940724223],
[21.382448674728483, 6.062801623535154], [21.393952251222306,
6.066063359517975], [21.40534079195119, 6.0692924781409685],
[21.416615447272786, 6.072489305577732], [21.427777356041165,
6.075654164740128], [21.43882764572186, 6.078787375310901], [21.449767432505748,
6.081889253775965], [21.460597821421796, 6.084960113456379],
[21.471319906448684, 6.088000264539989], [21.481934770625305,
6.091010014112762], [21.49244348616016, 6.093989666189808], [21.502847114539666,
6.096939521746084], [21.513146706635375, 6.099859878746797], [21.52334330281013,
6.102751032177502], [21.533437933023134, 6.1056132740739], [21.543431616934008,
6.108446893551335], [21.553325364005776, 6.111252176833995],
[21.563120173606826, 6.114029407283828], [21.572817035111864,
6.116778865429164], [21.58241692800185, 6.119500828993046]]
[290.61542216403615, 285.266000753095, 280.0230328282315, 274.8843999650728,
269.8480258958911, 264.91187567068596, 260.0739548349625, 255.33230862386984,

250.68502117237796, 246.13021474117076, 241.66604895794458, 237.29072007380464,
233.00246023445908, 228.79953676591649, 224.68025147439792, 220.6429399601805,
216.68597094509605, 212.8077456134118, 209.00669696582807, 205.28128918633118,
201.63001702164632, 198.05140517303872, 194.54400770021837, 191.10640743710718,
187.73721541923186, 184.43507032251227, 181.19863791321742, 178.02661050886752,
174.91770644986426, 171.87066958163504, 168.8842687470837, 165.95729728913983,
163.08857256320908, 160.27693545932436, 157.52124993380696, 154.8204025502473,
152.17330202962054, 149.57887880935422, 147.0360846111712, 144.54389201753202,
142.10129405650628, 139.70730379490496, 137.36095393950947, 135.06129644623638,
132.8074021370794, 130.59836032467464, 128.43327844433676, 126.31128169341758,
124.2315126778417, 122.1931310656758, 120.19531324759198, 118.23725200408805,
116.31815617932982, 114.4372503614843, 112.5937745694139, 110.7869839456057,
109.01614845521131, 107.28055259107572, 105.57949508463645, 103.91228862257533,
102.27825956910921, 100.67674769380709, 99.10710590482344, 97.5686999874406,
96.06090834781368, 94.5831217618153, 93.13474312887833, 91.7151872307368,
90.32388049496826, 88.96026076324155, 87.62377706417617, 86.3138893907222,
85.03006848196995, 83.77179560930188, 82.53856236679991, 81.32987046582376,
80.14523153367699, 78.98416691627996, 77.84620748476911, 76.73089344594536,
75.63777415649417, 74.56640794090309, 73.51636191300226, 72.48721180105666,
71.47854177633877, 70.48994428511277, 69.52101988396215, 68.57137707839446,
67.64063216465753, 66.72840907470399, 65.83433922424052, 64.95806136380126,
64.09922143278477, 63.25747241639548, 62.43247420543234, 61.62389345886739,
60.83140346915908, 60.05468403024594, 59.29342130816718, 58.54730771425779,
57.81604178086721, 57.09932803955109, 56.39687690168716, 55.70840454146673,
55.03363278121468, 54.372288978991634, 53.72410591843284, 53.088821700779164,
52.46617963905681, 51.855928154362715, 51.25782067421404, 50.671615532920306,
50.09707587393834, 49.533969554170106, 48.982069050165244, 48.44115136619009,
47.91099794412605, 47.39139457516108, 46.8821313132385, 46.38300239022819,
45.89380613278579, 45.41434488086649, 44.94442490786039, 44.4838563423171,
44.032453091228135, 43.59003276483583, 43.15641660293875, 42.731429402663395,
42.31489944767351, 41.90665843878794, 41.506541425979194, 41.114386741725355,
40.73003593568816, 40.35333371069109, 39.984127859971466, 39.62226920568118,
39.267611538611256, 38.92001155911603, 38.579328819212755, 38.24542566583356,
37.918167185206606, 37.59742114834414, 37.28305795761523, 36.97495059438182,
36.67297456767677, 36.377007863903145, 36.0869308975346, 35.80262646279681,
35.52397968631029, 35.250877980675845, 34.98321099898353, 34.7208705902269,
34.46375075560452, 34.21174760569112, 33.964759318461006, 33.72268609814676,
33.485430134916776, 33.25289556535506, 33.024988433727636, 32.80161665401959,
32.58268997272774, 32.3681199323936, 32.15781983586211, 31.951704711251583,
31.749691277620816, 31.5516979113193, 31.35764461300718, 31.167452975331475,
30.981046151245522, 30.798348822958868, 30.61928717150513, 30.443788846915314,
30.271782938984835, 30.103199948622176, 29.93797175976773, 29.776031611871485,
29.617314072918386, 29.461755012990448, 29.30929157835507, 29.159862166068937,
29.01340639908731, 28.869865101868605, 28.729180276464557, 28.59129507908606,
28.456153797135382, 28.323701826695526, 28.19388565046743, 28.06665281614626,
27.941951915228085, 27.81973256223818, 27.69994537437278, 27.5825419515459,
27.46747485683328, 27.354697597305435, 27.2441646052422, 27.135831219721023,
27.02965366857172, 26.92558905069027, 26.823595318704673, 26.72363126198558,

26.625656489995208, 26.529631415967433, 26.43551724091283, 26.343275937941797,
26.252870236899895, 26.16426360930872, 26.07742025360661, 25.992305080682975,
25.90888369970052, 25.82712240419962, 25.746988158479187, 25.668448584248583,
25.59147194754518, 25.51602714591217, 25.442083695831652, 25.369611720407736,
25.298581937294763, 25.228965646865735, 25.160734720616244, 25.093861589799115,
25.02831923428525, 24.964081171646107, 24.901121446453487, 24.839414619792194,
24.77893575898146, 24.719660427500873, 24.661564675116743, 24.604625028205056,
24.548818480266917, 24.494122482632743, 24.440514935351487, 24.387974178261132,
24.336478982236873, 24.28600854061349, 24.23654246077842, 24.188060755932064,
24.140543837012153, 24.093972504778748, 24.048327942056794, 24.003591706133,
23.95974572130409, 23.91677227157328, 23.87465399349211, 23.83337386914475,
23.792915219271904, 23.753261696531535, 23.71439727889369, 23.676306263166843,
23.638973258652964, 23.602383180928904, 23.566521245751552, 23.531372963084237,
23.496924131241993, 23.46316083115342, 23.4300694207366, 23.397636529387082,
23.365849052575417, 23.334694146552298, 23.30415922315905, 23.274231944741327,
23.244900219164105, 23.216152194925886, 23.187976256369993, 23.160361018991363,
23.133295324836574, 23.10676823799546, 23.080769040182492, 23.055287226405998,
23.030312500723657, 23.005834772082395, 22.98184415024109, 22.958330941774427,
22.935285646156252, 22.91269895192088, 22.890561732900792, 22.86886504453921,
22.847600120276017, 22.82675836800566, 22.80633136660549, 22.786310862533174,
22.7666887664919, 22.74745715016185, 22.728608242996764, 22.710134429084263,
22.69202824406862, 22.674282372134794, 22.65688964305245, 22.63984302927884,
22.62313564311933, 22.60676073394439, 22.590711685462036, 22.57498201304448,
22.55956536110803, 22.54445550054512, 22.52964632620741, 22.51513185443902,
22.500906220658816, 22.486963676990843, 22.473298589941855, 22.45990543812515,
22.4467788100296, 22.43391340183315, 22.421304015259807, 22.408945555479274,
22.396833029048377, 22.38496154189345, 22.373326297332905, 22.361922594139113,
22.350745824638885, 22.33979147285171, 22.3290551126651, 22.318532406046195,
22.30821910128902, 22.2981110312965, 22.288204111896835]

[[0.0, 0.0, 0.0], [1.1266403162055347, 0.16558879839386795, 0.3194487610908542],
[2.1969486166007925, 0.31791490653760923, 0.6203419768034926],
[3.213741501976287, 0.4579309167188106, 0.9038143161328905], [4.179694743083007,
0.5865240913083674, 1.1709288546329089], [5.097350322134392, 0.7045207460803069,
1.4226816732043024], [5.969123122233206, 0.8126903426266912, 1.660006158565603],
[6.797307282327081, 0.9117493090664497, 1.8837770248607129], [7.584082234416263,
1.0023646069835892, 2.094814074584857], [8.331518438900986, 1.0851570613498152,
2.293885715821682], [9.041582833161472, 1.1607044690837796, 2.4817122516732733],
[9.716144007708934, 1.2295445008688068, 2.6589689567266097],
[10.356977123529022, 1.2921774098883037, 2.8262889544297067],
[10.965768583558106, 1.3490685602386843, 2.9842659083439522],
[11.544120470585735, 1.4006507869393823, 3.133456539391776],
[12.093554763261984, 1.4473265986744792, 3.274382980426888],
[12.615517341304422, 1.489470233667067, 3.4075349787142746],
[13.111381790444735, 1.5274295784022396, 3.5333719562155217],
[13.582453017128035, 1.5615279582744537, 3.652324936928694],
[14.029970682477169, 1.592065808636927, 3.7647983499279656],
[14.455112464558846, 1.6193222341719962, 3.8711717161836874],

[14.85899715753644, 1.6435564639793578, 3.971801226716003], [15.242687615865153,
1.6650092092914441, 4.067021219142095], [15.607193551277431, 1.6839039302695875,
4.157145559216348], [15.953474189919095, 1.7004480179090002, 4.242468933532076],
[16.282440796628677, 1.714833896682967, 4.323268059151009], [16.59495907300278,
1.7272400531851773, 4.399802815550566], [16.891851435558177, 1.7378319956820956,
4.472317303927436], [17.173899179985806, 1.7467631491630764, 4.541040838567411],
[17.44184453719205, 1.7541756901730707, 4.606188874684364], [17.696392626537985,
1.7602013254298454, 4.667963876844288], [17.93821331141662, 1.7649620179633358,
4.726556131822063], [18.167942962051328, 1.7685706642678474, 4.782144509487957],
[18.386186130154297, 1.7711317257271755, 4.834897175086549],
[18.593517139852118, 1.7727418173572544, 4.8849722560517606],
[18.79048159906505, 1.7734902567096724, 4.932518466297042], [18.977597835317333,
1.7734595755913776, 4.977675690728401], [19.155358259757, 1.7727259970802713,
5.0205755325491985], [19.32423066297469, 1.771359880152324, 5.061341825758484],
[19.48465944603149, 1.7694261340826027, 5.1000911150884445],
[19.637066789935453, 1.7669846046394484, 5.136933105480513], [19.78185376664422,
1.7640904339573291, 5.171971083063231], [19.919401394517543, 1.7607943958489942,
5.205302309467367], [20.050071640997203, 1.7571432082008898, 5.237018391194567],
[20.17420837515288, 1.7531798239868268, 5.267205625644356], [20.292138272600774,
1.7489437023331076, 5.295945325300101], [20.404171675176272, 1.7444710609732468,
5.323314121477206], [20.510603407622995, 1.7397951113416281, 5.349384248945705],
[20.61171355344738, 1.7349462774724997, 5.374223812654383], [20.70776819198055,
1.7299523997932484, 5.397897037703973], [20.799020098587057, 1.7248389248285467,
5.420464503642605], [20.885709409863242, 1.719629081764393, 5.441983364087193],
[20.968064255575616, 1.7143440467579607, 5.462507552609438], [21.04630135900237,
1.7090030958202285, 5.482087975764368], [21.12062660725779, 1.7036237470433258,
5.500772694082562], [21.191235593100437, 1.6982218928931176, 5.518607091794075],
[21.25831412965095, 1.692811923239546, 5.535634036002475], [21.32203873937394,
1.6874068397524147, 5.551894025980949], [21.38257711861078, 1.682018362248436,
5.567425333219097], [21.440088578885778, 1.6766570275362602,
5.5822641328084135], [21.494724466147026, 1.6713322812697033,
5.596444626716563], [21.54662855904521, 1.6660525632852916, 5.609999159465071],
[21.595937447298486, 1.6608253868684109, 5.62295832669191], [21.642780891139097,
1.655657412362625, 5.635351077049437], [21.687282162787678, 1.6505545155089674,
5.6472048078591675], [21.72955837085383, 1.6455218508760976, 5.658545454917746],
[21.769720768516674, 1.6405639107180208, 5.669397576823139],
[21.807875046296378, 1.6356845795734718, 5.679784434166366],
[21.844121610187095, 1.6308871848999762, 5.689728063911921],
[21.878555845883277, 1.626174544015908, 5.699249349269301], [21.91126836979465,
1.621549007605483, 5.7083680853386864], [21.942345267510454, 1.6170125000244593,
5.717103040795684], [21.97186832034047, 1.6125665566283012, 5.7254720158630725],
[21.999915220528983, 1.6082123583296013, 5.733491896801672], [22.02655977570807,
1.6039507635776011, 5.741178707137579], [22.051872103128204, 1.5997823379396143,
5.748547655829182], [22.07591881417733, 1.5957073814519958, 5.755613182564367],
[22.098763189674003, 1.5917259538969473, 5.762389000366197], [22.12046534639584,
1.5878378981508545, 5.768888135674009], [22.141082395281586, 1.5840428617399611,
5.775122966056229]]
[268.81364342400354, 244.271596513659, 222.1743175984097, 202.2764693142609,

```
184.3575454514381, 168.2193325882372, 153.68363422287428, 140.59022989738375,
128.79504473266235, 118.16850740271391, 108.59407690375427, 99.96692055095055,
92.19272748960256, 85.18664366289273, 78.87231565658351, 73.1810321614865,
68.05095297428693, 63.42641651161476, 59.257317754665, 55.49854938425256,
52.10949961968176, 49.053600948785906, 46.297924539470095, 43.81281566266539,
41.571565939540896, 39.55011865815863, 37.72680379187212, 36.08209969846456,
34.598418789576975, 33.259914738211606, 32.05230904142275, 30.962734978769987,
29.97959720742783, 29.092445414449745, 28.291860607744383, 27.569352771776035,
26.917268743585073, 26.328709280986956, 25.79745439913579, 25.317896145271863,
24.88497806552044, 24.49414069306419, 24.14127245476279, 23.822665454135766,
23.53497564327088, 23.275186945307997, 23.040578933248863, 22.828697710468937,
22.63732967391628, 22.464477872986418, 22.30834070582933, 22.16729272070564,
22.03986731326036, 21.924741131489643, 21.820720018977543, 21.726726343889627,
21.641787576420604, 21.565025991075537, 21.49564938247455, 21.432942694446723,
21.376260472145542, 21.325020055887954, 21.278695443491483, 21.23681175515014,
21.19894024143066, 21.16469378085921, 21.133722818870133, 21.105711704661992,
21.080375386804647, 21.057456432312232, 21.03672233738304, 21.017963101147657,
21.00098903659502, 20.98562879539393, 20.971727585623093, 20.959145563490647,
20.947756381987325, 20.937445881096597, 20.928110905697682]
```

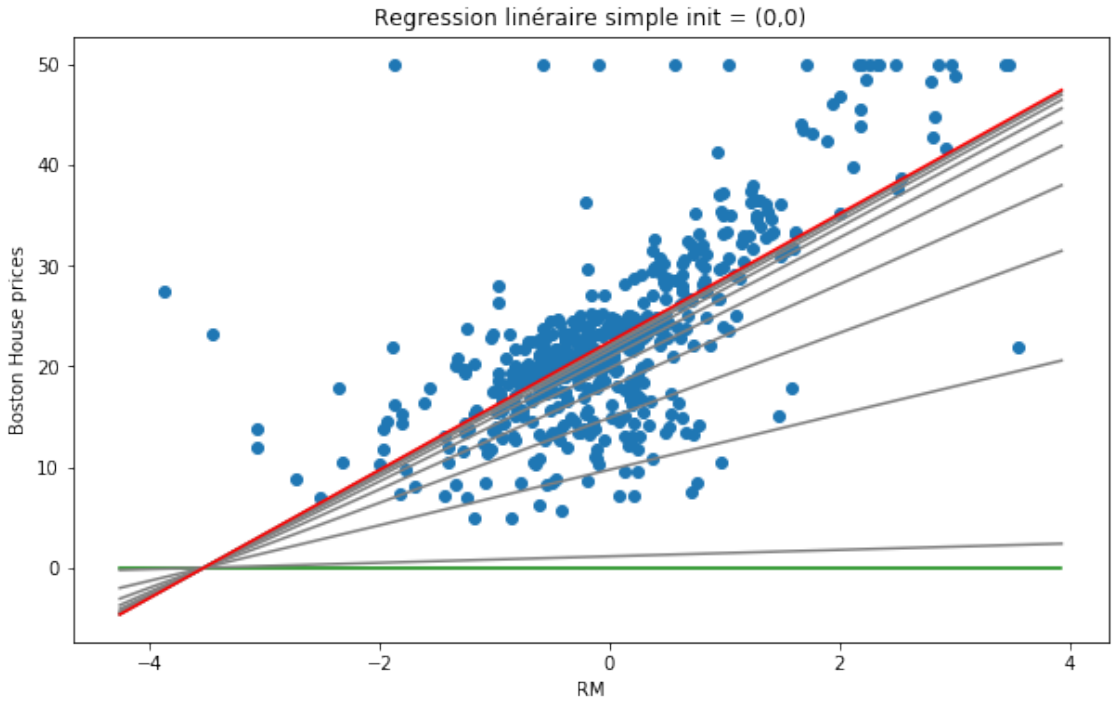
1.4 Test avec plusieurs initialisations : visualisation de la fonction cout et des valeurs des paramètres au cours des itérations

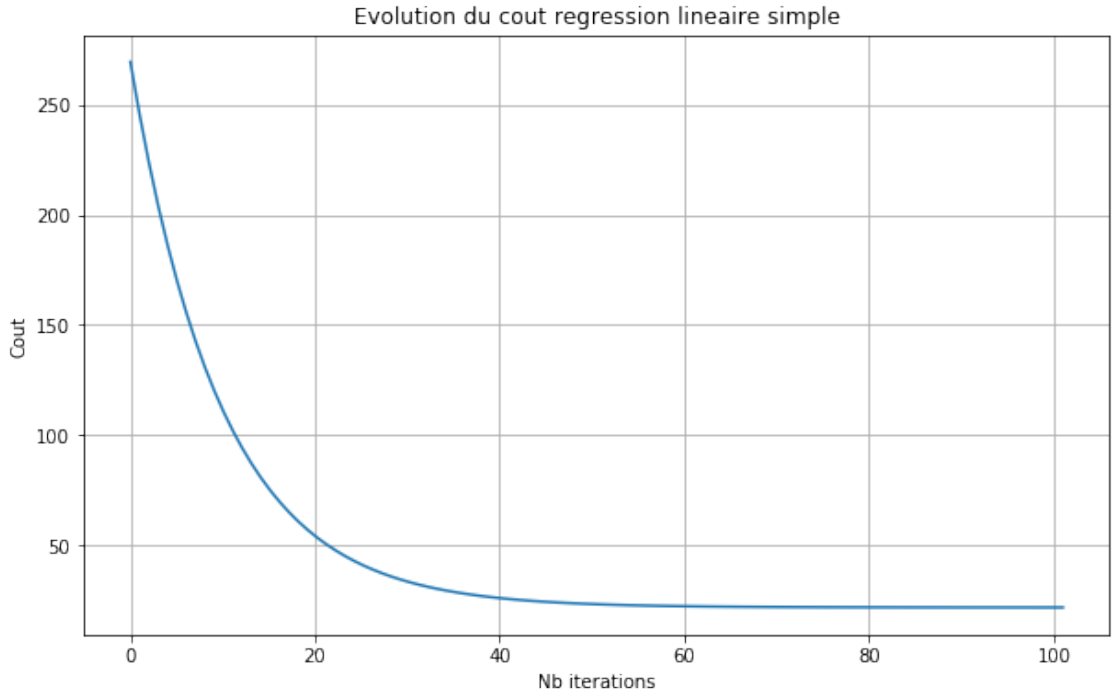
[19]:
```python
def display(title,data_X,data_Y,data_b,data_c=None):
    print("\nBetas :",data_b[-1])
    plt.figure(1,figsize=(10,6))
    plt.title(title)
    plt.scatter(standardisation(data_X)[:,1],data_Y)
    x_lim = plt.xlim()
    x = np.linspace(x_lim[0],x_lim[1],100)
    y = data_b[0][0] + (x*data_b[0][1])
    plt.plot(x,y,color='green')
    for i in range(1,len(data_b)-1,10):
        y = data_b[i][0] + (x*data_b[i][1])
        plt.plot(x,y,color='gray')
    y = data_b[-1][0] + (x*data_b[-1][1])
    plt.plot(x,y,color='red')
    plt.xlabel(names[k-1])
    plt.ylabel("Boston House prices")
    plt.show()
    if data_c != None:
        displayCout(data_c,"regression lineaire simple")

def display3D(title,data_X,data_Y,data_b,data_c=None):
    print("\nBetas :",data_b[-1])
    fig = plt.figure(1,figsize=(12,8))
    ax = fig.add_subplot(111, projection='3d')
```

```
    ax.set_title(title)
    xx = standardisation(data_X)
    ax.scatter(xx[:,1],xx[:,2],data_Y,c='r', marker='o')
    x_lim = plt.xlim()
    y_lim = plt.ylim()
    x = np.arange(x_lim[0], x_lim[1], 0.05)
    y = np.arange(y_lim[0], y_lim[1], 0.05)
    x, y = np.meshgrid(x, y)
    z = data_b[-1][0] + (x*data_b[-1][1]) + (y*data_b[-1][2])
    z = z.reshape(x.shape)
    ax.plot_wireframe(x,y,z,color='gray')
    ax.set_xlabel(names[kk-1])
    ax.set_ylabel(names[k-1])
    ax.set_zlabel("Boston House prices")
    plt.show()
    if data_c != None:
        displayCout(data_c,"regression lineaire multiple")

def displayCout(data_c,title):
    plt.figure(1,figsize=(10,6))
    plt.grid()
    plt.title("Evolution du cout "+title)
    x = np.linspace(0, len(data_c), len(data_c))
    plt.plot(x, data_c)
    plt.xlabel("Nb iterations")
    plt.ylabel("Cout")
    plt.show()
```

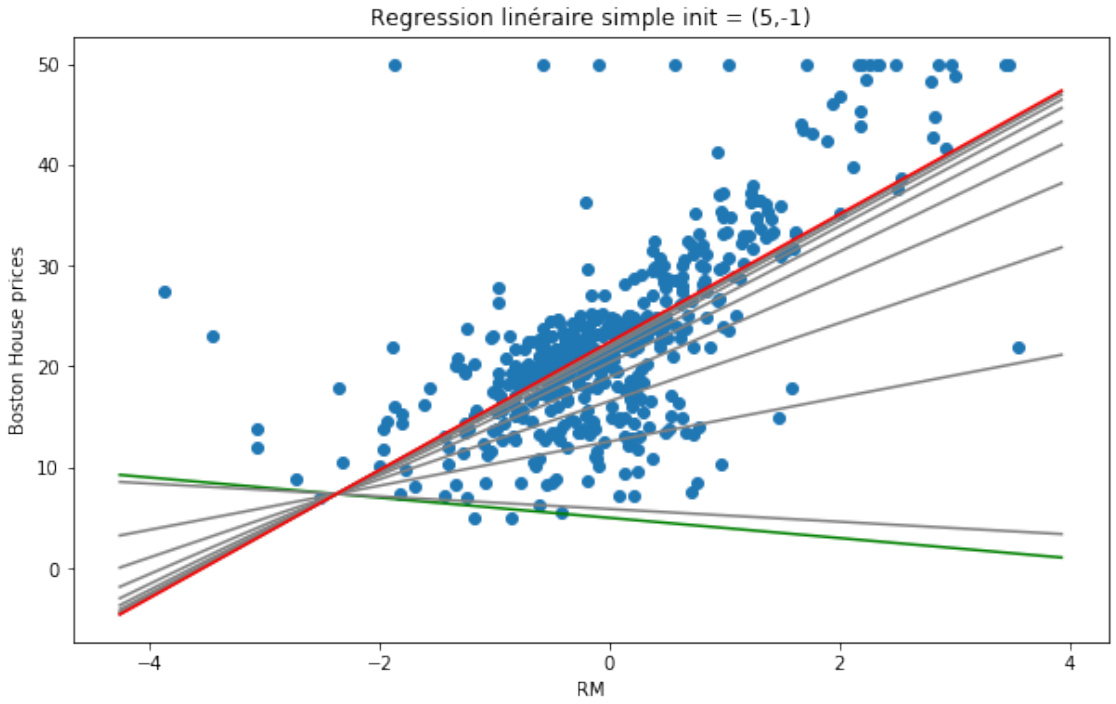
[20]:
```
# -- Test 1  - reg simple
b1a, c = grad_descent((0,0),x1,Y,0.05,0.001,500)
display("Regression linéraire simple init = (0,0)",x1,Y,b1a,c)
b, c = grad_descent((5,-1),x1,Y,0.05,0.001,500)
display("Regression linéraire simple init = (5,-1)",x1,Y,b,c)
#-- Test 2 - reg multiple
b1b, c = grad_descent((0,0,0),x2,Y,0.05,0.001,500)
display3D("Regression linéraire multiple init = (0,0,0)",x2,Y,b1b,c)
b, c = grad_descent((12,1,1),x2,Y,0.05,0.001,500)
display3D("Regression linéraire multiple init = (12,1,1)",x2,Y,b,c)
```

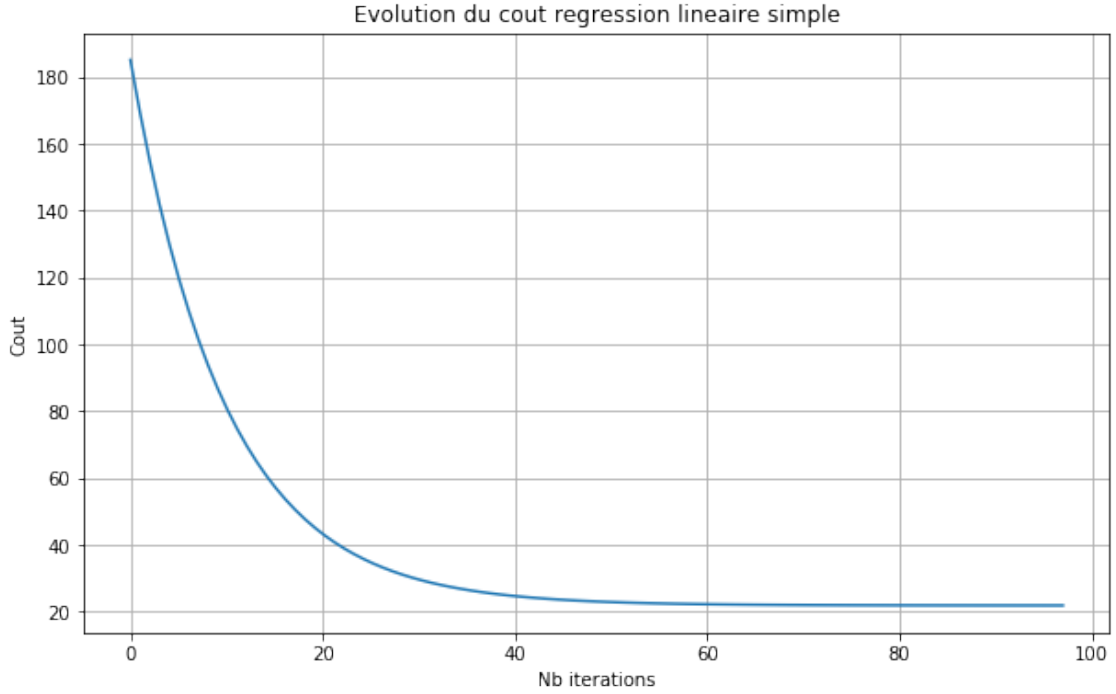
```
Betas : [22.406070492765547, 6.353040413053022]
```



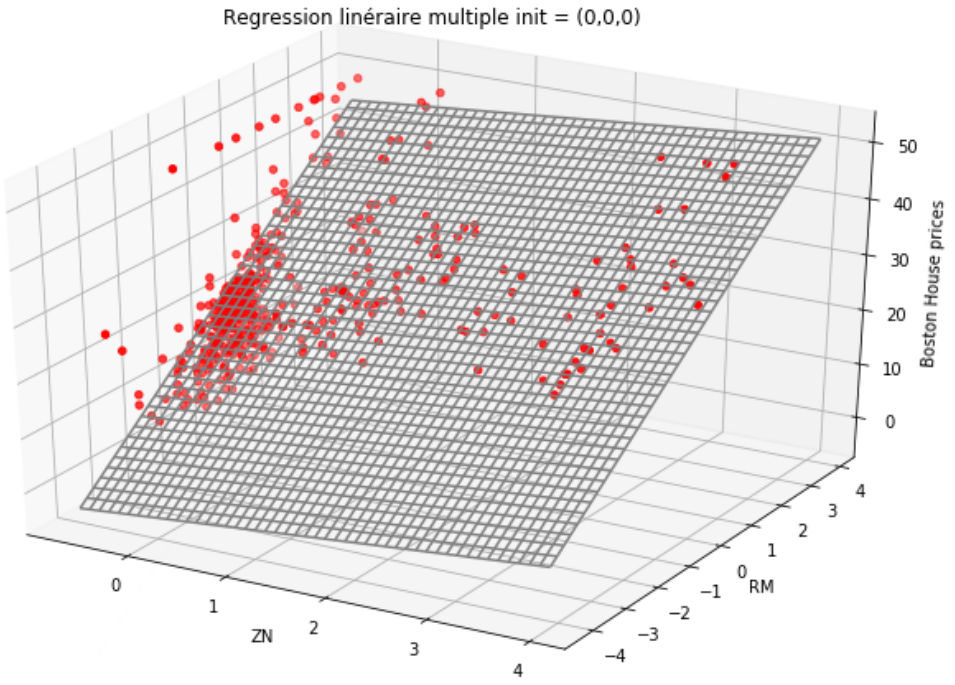


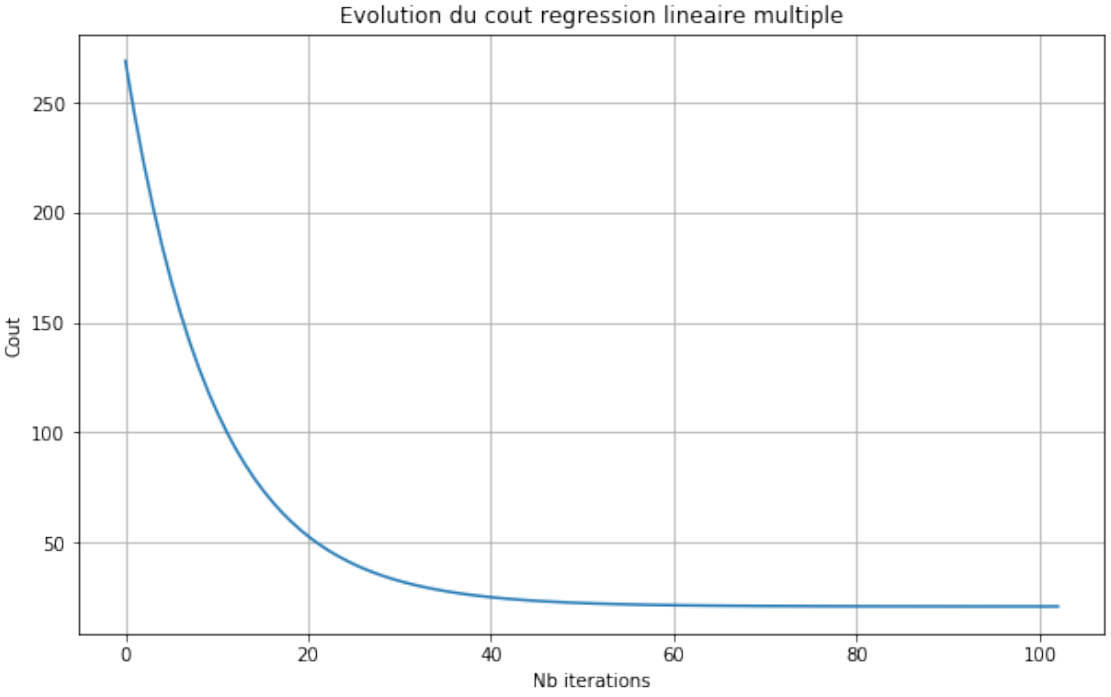

```
Betas : [22.411735040067686, 6.337951289804988]
```



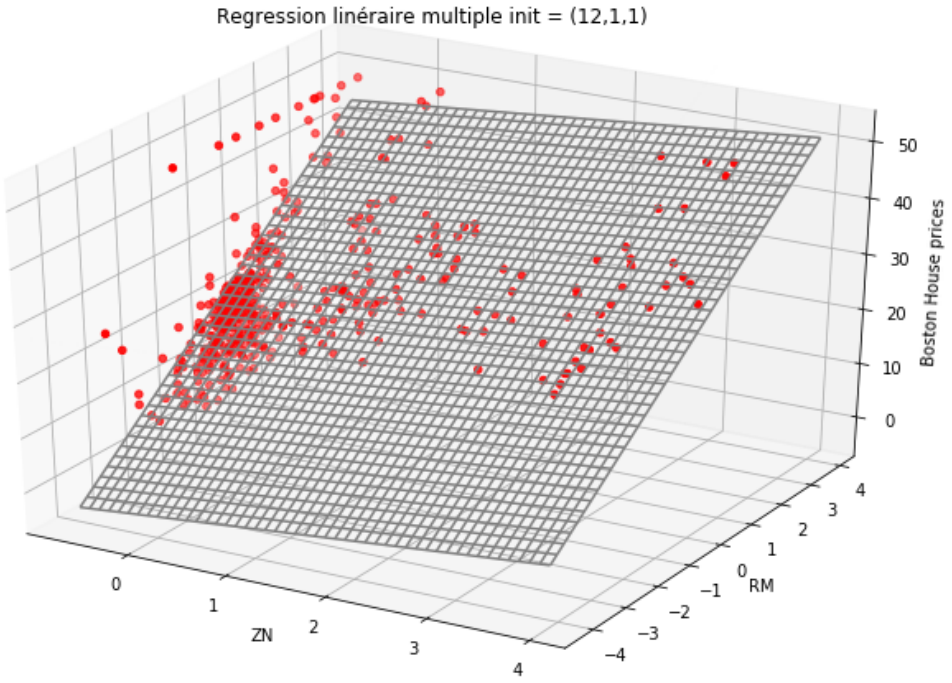


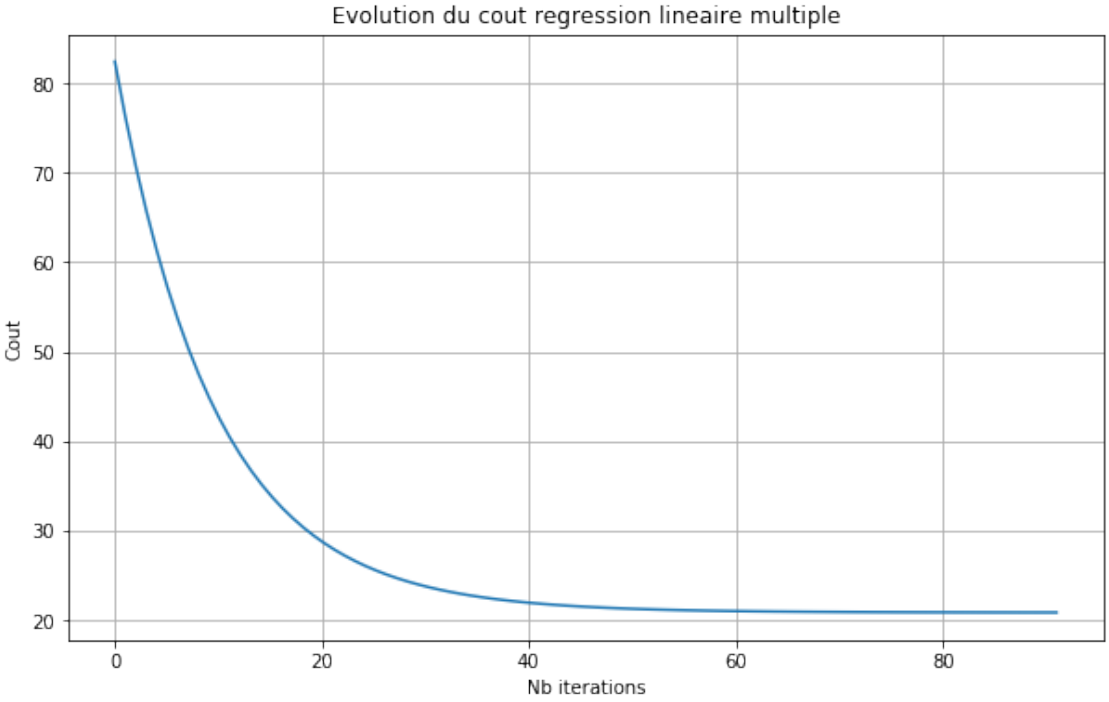

```
Betas : [22.412407284332872, 1.5199296820757724, 5.866689898479008]
```


Regression linéraire multiple init = (0,0,0)
Boston House prices
ZN
RM


Evolution du cout regression lineaire multiple
Cout
Nb iterations

```
Betas : [22.433861705279174, 1.5475207544295067, 5.835166699467882]
```





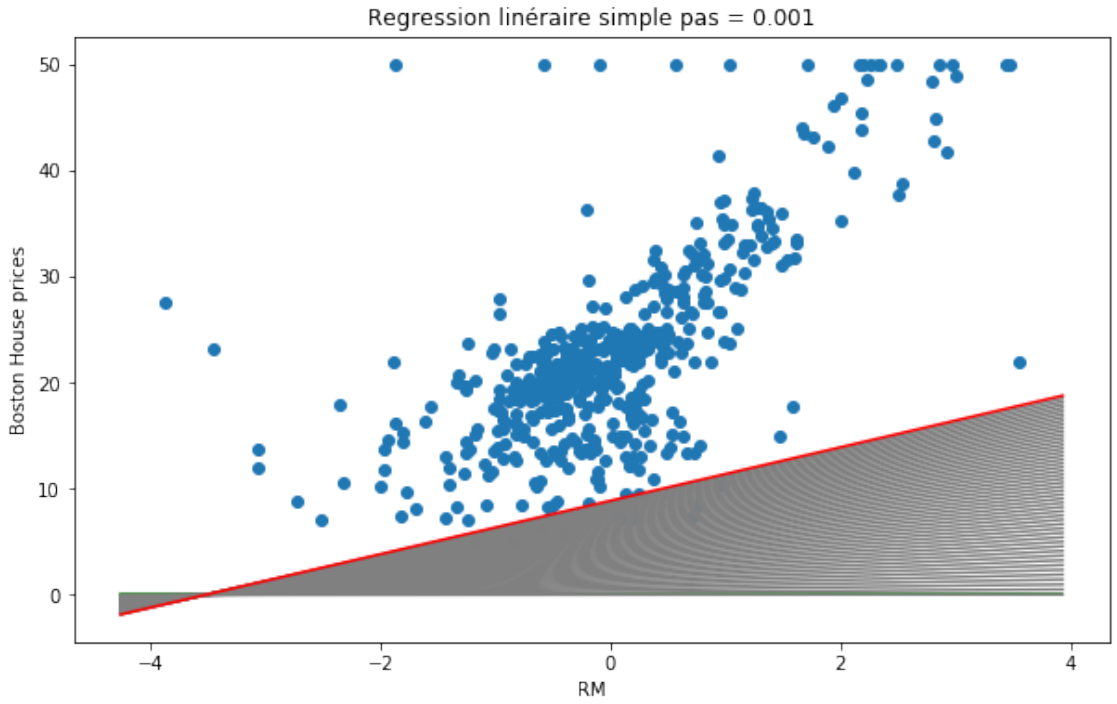
Commentaires:

L'initialisation des Betas ne change pas le résultat final, elle détermine juste le nombre d'itérations qu'il va falloir pour y arriver (chance ou pas).

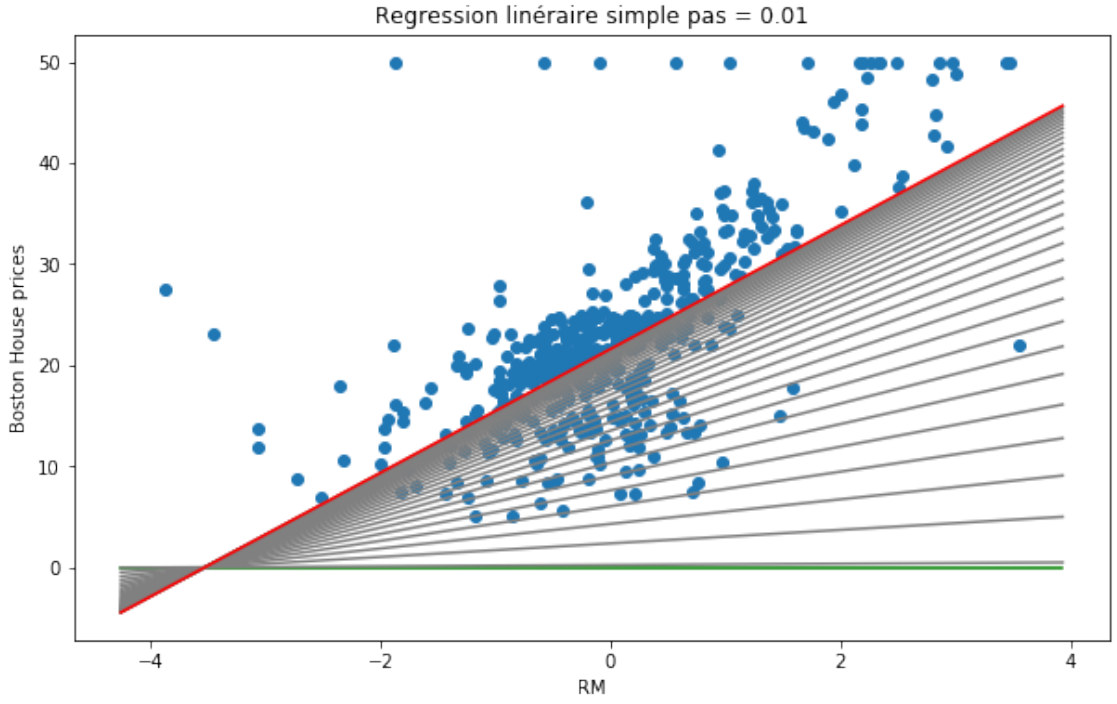
1.5 Test avec plusieurs pas de descente : visualisation de la fonction cout et des valeurs des paramètres au cours des itérations

[114]:
```
#-- Test 1  - reg simple
b, c = grad_descent((0,0),x1,Y,0.001,0.01,500)
display("Regression linéraire simple pas = 0.001",x1,Y,b)
b, c = grad_descent((0,0),x1,Y,0.01,0.01,500)
display("Regression linéraire simple pas = 0.01",x1,Y,b)
b, c = grad_descent((0,0),x1,Y,0.1,0.01,500)
display("Regression linéraire simple pas = 0.01",x1,Y,b)


#-- Test 2 - reg multiple
b, c = grad_descent((0,0,0),x2,Y,0.001,0.01,500)
display3D("Regression linéraire multiple pas = 0.001",x2,Y,b)
b, c = grad_descent((12,1,1),x2,Y,0.01,0.01,500)
display3D("Regression linéraire multiple pas = 0.01",x2,Y,b)
b, c = grad_descent((12,1,1),x2,Y,0.1,0.01,500)
display3D("Regression linéraire multiple pas = 0.1",x2,Y,b)
```

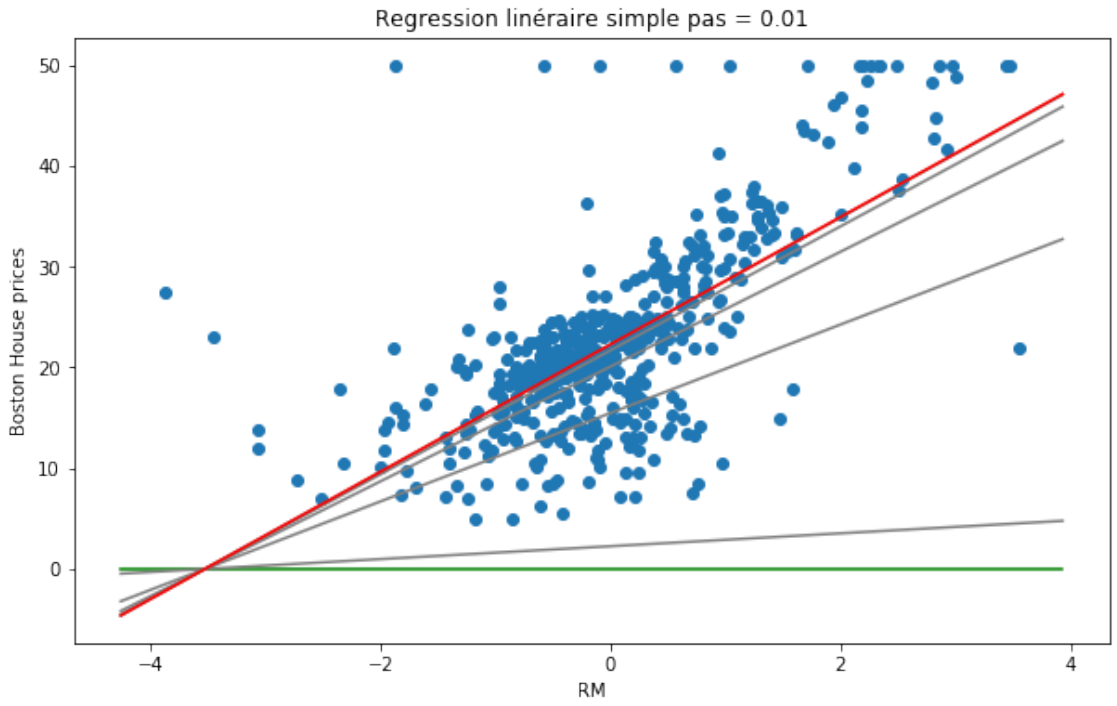
```
Betas : [8.869387000535, 2.5148351680674854]
```



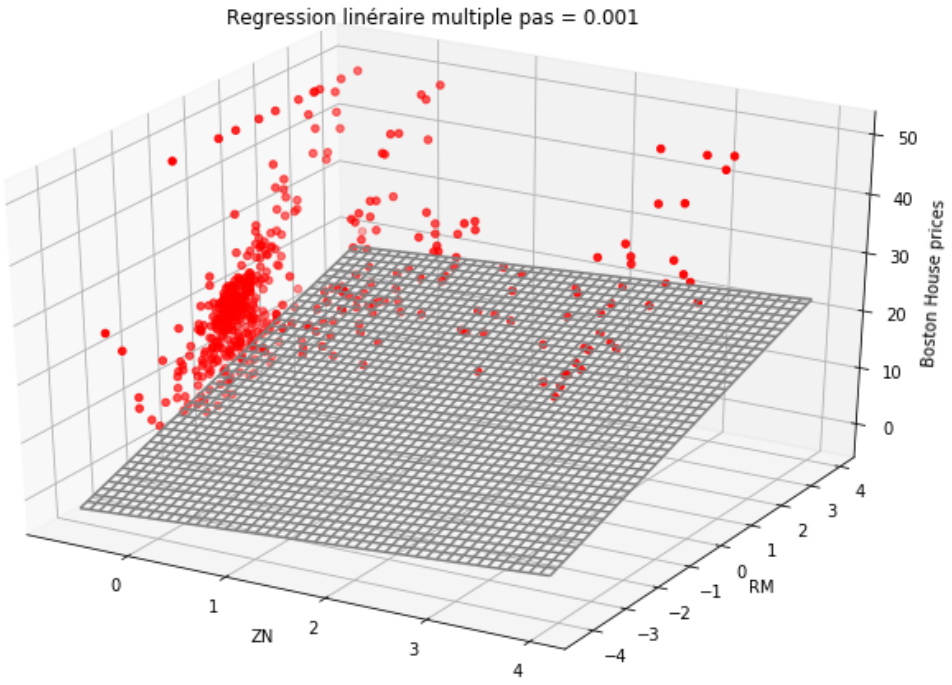

```
Betas : [21.58241692800185, 6.119500828993046]
```



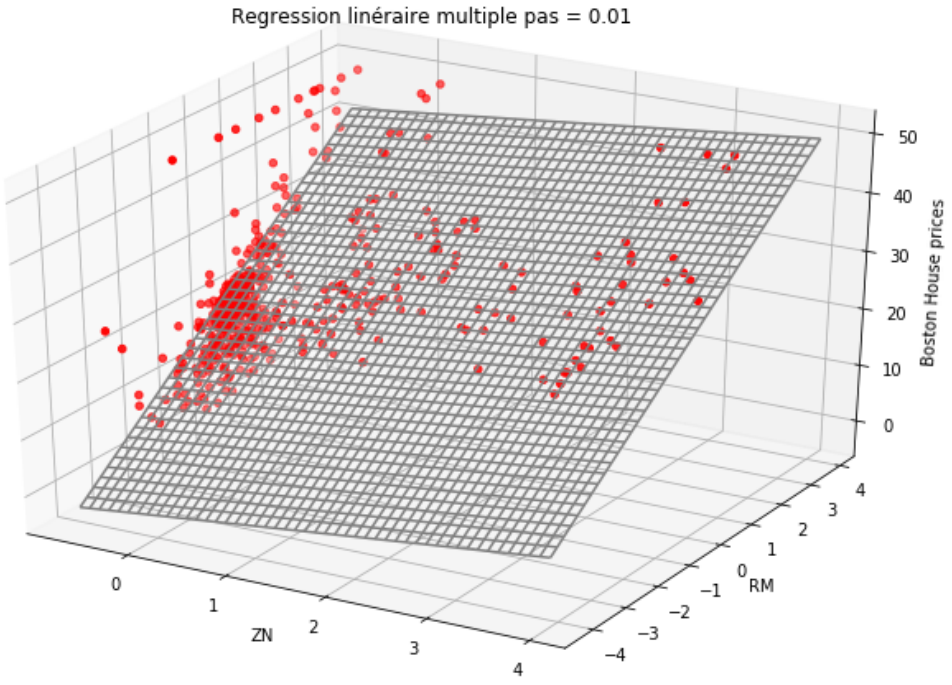
```
Betas : [22.263031958289528, 6.312483118973122]
```



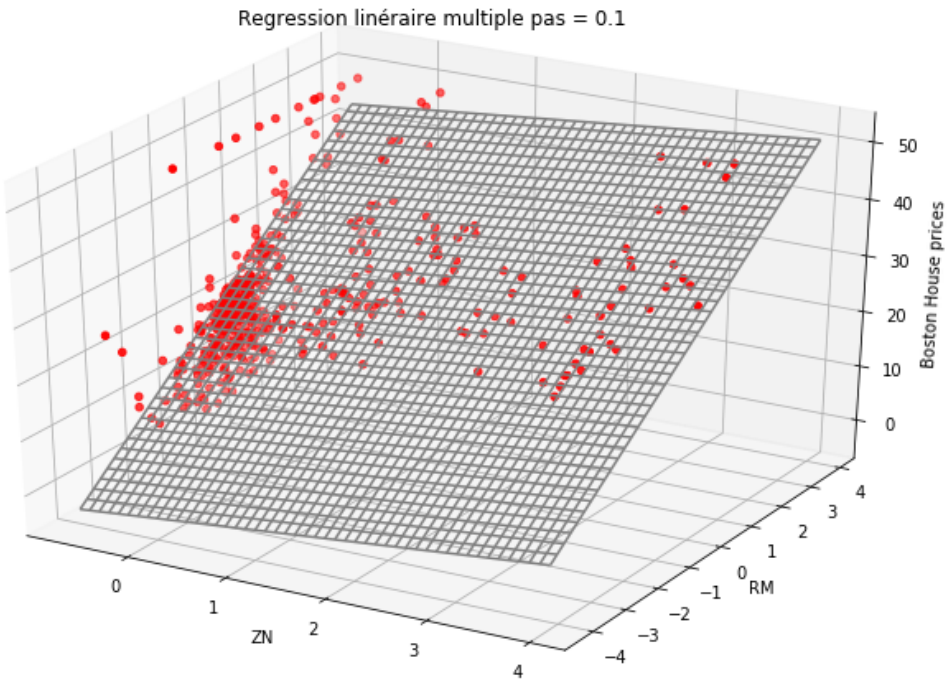

```
Betas : [8.869387000535006, 1.128220382360806, 2.430458690058276]
```



Betas : [21.652898531094287, 1.7637290522481617, 5.423552045352301]



```
Betas : [22.295519054911704, 1.6154075357782463, 5.744388741165023]
```



Commentaires:

On remarque qu'un pas trop petit demande trop d'itérations avant d'arriver au résultat (sortie avec ITE_MAX). Un pas trop grand lui peut créer une divergence (va et vient entre 2 valeurs).

1.6 Commentaires

pour le choix du critère d'arrêt:

Le critère d'arrêt dépend donc d'un nombre d'itérations max qui est donné et de la différence entre le nouveau et l'ancien cout qui ne doit pas être inférieure à une valeur donnée aussi. Donc on remarque donc que le résultat est fortément dépendant des paramètres au départ.

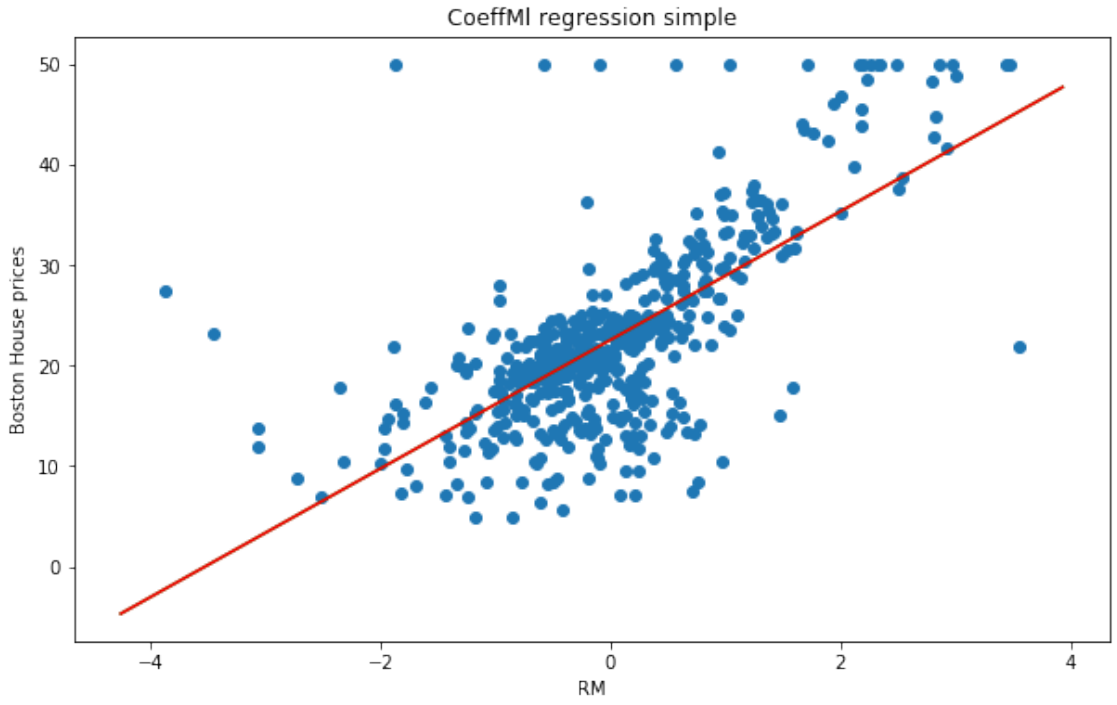
pour la standardisation des données:

Le fait de standardiser les données permet de gagner en temps d'exécution car les valeurs calculées des Betas seront relativement proches (hypothèse).

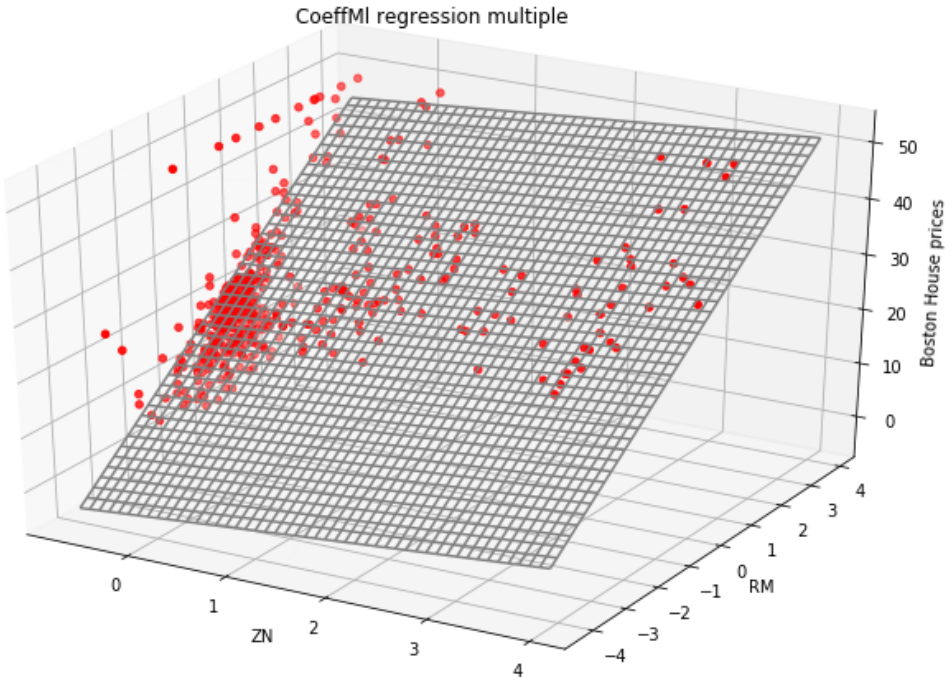
1.7 Comparaison avec les résultats du maximum de vraisemblance

[115]:
```
#-- Pour comparer avec le maximum de vraisemblance
#(X^T X)^-1 X^T Y
def coef_ml(x,y):
    x_standard = standardisation(x)
    trans = np.transpose(x_standard)
    return np.dot(np.dot(np.linalg.inv(np.dot(trans,x_standard)),trans),y)

#-- Test 1  - reg simple
maxi1 = coef_ml(x1,Y)
print("Betas trouvés avec la regression linéaire :",b1a[-1])
print("Différence : ",np.sum(np.abs(maxi1-b1a[-1])))
display("CoeffMl regression simple",x1,Y,[maxi1])

#-- Test 2 - reg multiple
maxi2 = coef_ml(x2,Y)
print("Betas trouvés avec la regression linéaire :",b1a[-1])
print("Différence : ",np.sum(np.abs(maxi2-b1b[-1])))
display3D("CoeffMl regression multiple",x2,Y,[maxi2])
#-- (ou bien utiliser la fonction native de sklearn)
```

```
Betas trouvés avec la regression linéaire : [22.406070492765547,
6.353040413053022]
Différence :  0.16267064010944043

Betas : [22.53280632  6.38897522]
```



```
Betas trouvés avec la regression linéaire : [22.406070492765547,
6.353040413053022]
Différence :  0.24625096714650652

Betas : [22.53280632  1.46065307  5.93326521]
```



Commentaires:

La différence entre les Betas de la régression linéaire et les Betas du maximum de vraisemblance est relativement faible. Je pense qu'elle est acceptable.

AlgoRegLog

May 11, 2020

1 TD 5 - La régression logistique - algo. de descente du gradient

1.1 Packages utiles

[17]:
```
from sklearn import datasets # donnees
import os # rep de travail
import pandas as pd # data analysis
from scipy import stats # stat desc
import matplotlib.pyplot as plt # graphiques
import numpy as np # maths
import sklearn.preprocessing # standardisation des donnees
from collections import Counter # freq table

from sklearn.linear_model import LogisticRegression    #Logistic Regression
from sklearn.model_selection import train_test_split # train and test samples
from sklearn.metrics import confusion_matrix # confusion matrix
from scipy import optimize # algos d'optim
from mpl_toolkits.mplot3d import Axes3D
from sklearn.model_selection import train_test_split
```

1.2 Les données

[2]:
```
#-- Import des données
breast_cancer = datasets.load_breast_cancer()
print(breast_cancer.DESCR)
#Data Set Characteristics:
#    :Number of Instances: 569
#
#    :Number of Attributes: 30 numeric, predictive attributes and the class
#
#    :Attribute Information:
#        - radius (mean of distances from center to points on the perimeter)
#        - texture (standard deviation of gray-scale values)
#        - perimeter
#        - area
#        - smoothness (local variation in radius lengths)
```

```
#         - compactness (perimeter^2 / area - 1.0)
#         - concavity (severity of concave portions of the contour)
#         - concave points (number of concave portions of the contour)
#         - symmetry
#         - fractal dimension ("coastline approximation" - 1)
#
#         The mean, standard error, and "worst" or largest (mean of the three
#         largest values) of these features were computed for each image,
#         resulting in 30 features.  For instance, field 3 is Mean Radius, field
#         13 is Radius SE, field 23 is Worst Radius.
#
#         - class:
#                 - WDBC-Malignant
#                 - WDBC-Benign
```

```
.. _breast_cancer_dataset:

Breast cancer wisconsin (diagnostic) dataset
--------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 569

    :Number of Attributes: 30 numeric, predictive attributes and the class

    :Attribute Information:
        - radius (mean of distances from center to points on the perimeter)
        - texture (standard deviation of gray-scale values)
        - perimeter
        - area
        - smoothness (local variation in radius lengths)
        - compactness (perimeter^2 / area - 1.0)
        - concavity (severity of concave portions of the contour)
        - concave points (number of concave portions of the contour)
        - symmetry
        - fractal dimension ("coastline approximation" - 1)

        The mean, standard error, and "worst" or largest (mean of the three
        largest values) of these features were computed for each image,
        resulting in 30 features.  For instance, field 3 is Mean Radius, field
        13 is Radius SE, field 23 is Worst Radius.

        - class:
                - WDBC-Malignant
                - WDBC-Benign
```

:Summary Statistics:

	Min	Max
radius (mean):	6.981	28.11
texture (mean):	9.71	39.28
perimeter (mean):	43.79	188.5
area (mean):	143.5	2501.0
smoothness (mean):	0.053	0.163
compactness (mean):	0.019	0.345
concavity (mean):	0.0	0.427
concave points (mean):	0.0	0.201
symmetry (mean):	0.106	0.304
fractal dimension (mean):	0.05	0.097
radius (standard error):	0.112	2.873
texture (standard error):	0.36	4.885
perimeter (standard error):	0.757	21.98
area (standard error):	6.802	542.2
smoothness (standard error):	0.002	0.031
compactness (standard error):	0.002	0.135
concavity (standard error):	0.0	0.396
concave points (standard error):	0.0	0.053
symmetry (standard error):	0.008	0.079
fractal dimension (standard error):	0.001	0.03
radius (worst):	7.93	36.04
texture (worst):	12.02	49.54
perimeter (worst):	50.41	251.2
area (worst):	185.2	4254.0
smoothness (worst):	0.071	0.223
compactness (worst):	0.027	1.058
concavity (worst):	0.0	1.252
concave points (worst):	0.0	0.291
symmetry (worst):	0.156	0.664
fractal dimension (worst):	0.055	0.208

:Missing Attribute Values: None

:Class Distribution: 212 - Malignant, 357 - Benign

:Creator: Dr. William H. Wolberg, W. Nick Street, Olvi L. Mangasarian

:Donor: Nick Street

:Date: November, 1995

This is a copy of UCI ML Breast Cancer Wisconsin (Diagnostic) datasets.

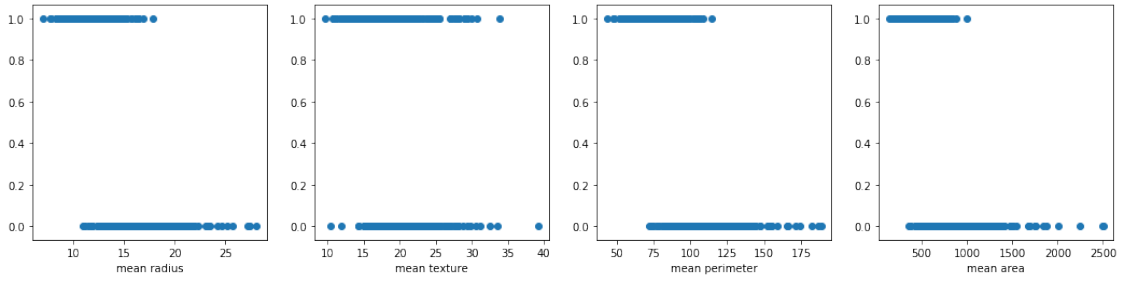
```
https://goo.gl/U2Uwz2

Features are computed from a digitized image of a fine needle
aspirate (FNA) of a breast mass.  They describe
characteristics of the cell nuclei present in the image.

Separating plane described above was obtained using
Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree
Construction Via Linear Programming." Proceedings of the 4th
Midwest Artificial Intelligence and Cognitive Science Society,
pp. 97-101, 1992], a classification method which uses linear
programming to construct a decision tree.  Relevant features
were selected using an exhaustive search in the space of 1-4
features and 1-3 separating planes.

The actual linear program used to obtain the separating plane
in the 3-dimensional space is that described in:
[K. P. Bennett and O. L. Mangasarian: "Robust Linear
Programming Discrimination of Two Linearly Inseparable Sets",
Optimization Methods and Software 1, 1992, 23-34].

This database is also available through the UW CS ftp server:

ftp ftp.cs.wisc.edu
cd math-prog/cpo-dataset/machine-learn/WDBC/

.. topic:: References

   - W.N. Street, W.H. Wolberg and O.L. Mangasarian. Nuclear feature extraction
     for breast tumor diagnosis. IS&T/SPIE 1993 International Symposium on
     Electronic Imaging: Science and Technology, volume 1905, pages 861-870,
     San Jose, CA, 1993.
   - O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer diagnosis and
     prognosis via linear programming. Operations Research, 43(4), pages
570-577,
     July-August 1995.
   - W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Machine learning
techniques
     to diagnose breast cancer from fine-needle aspirates. Cancer Letters 77
(1994)
     163-171.
```

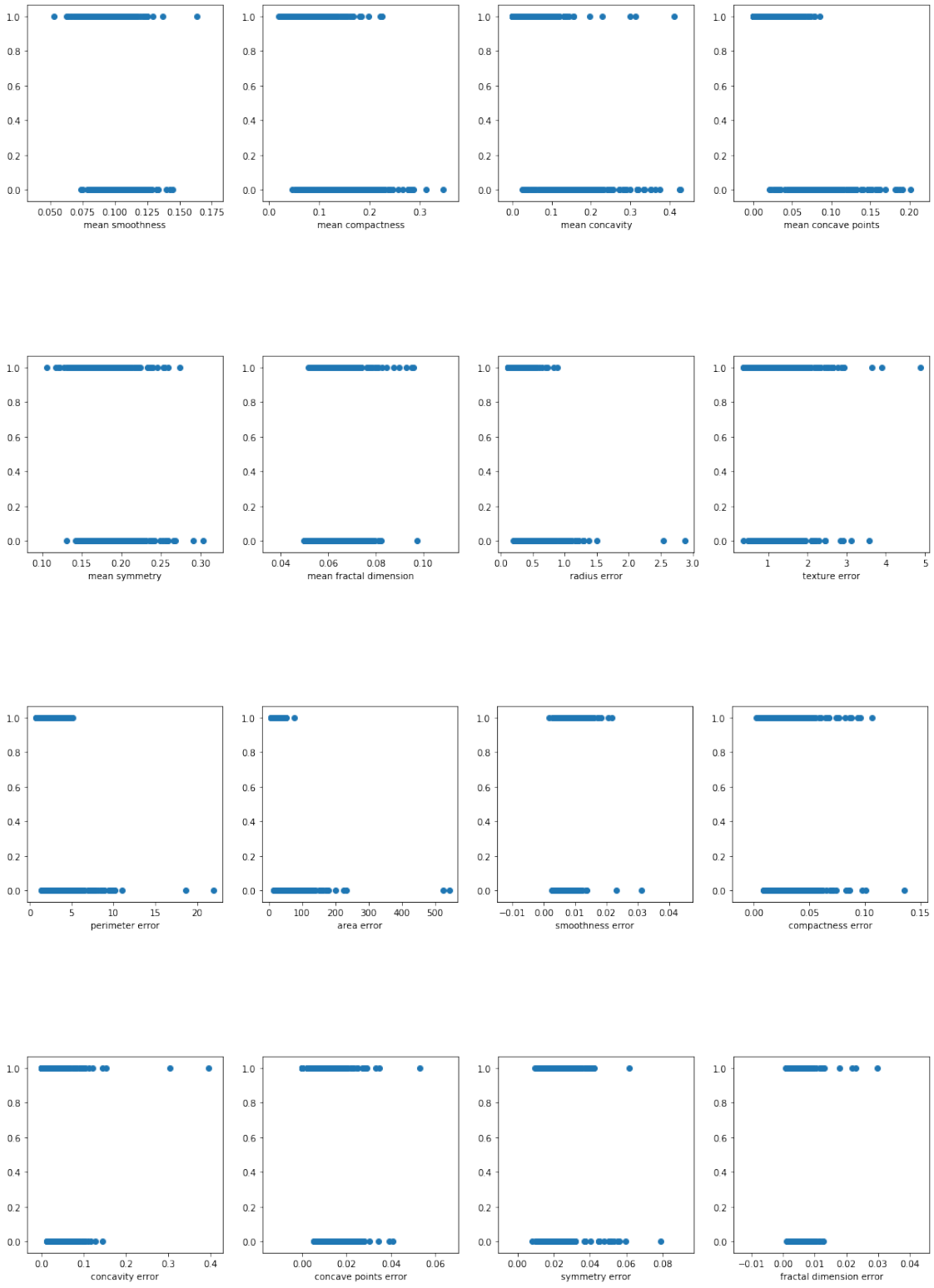
[3]:
```
#-- Stat. descriptives
dat=breast_cancer.data
Y=breast_cancer.target
names = breast_cancer.feature_names
#%% Stat descriptives
```

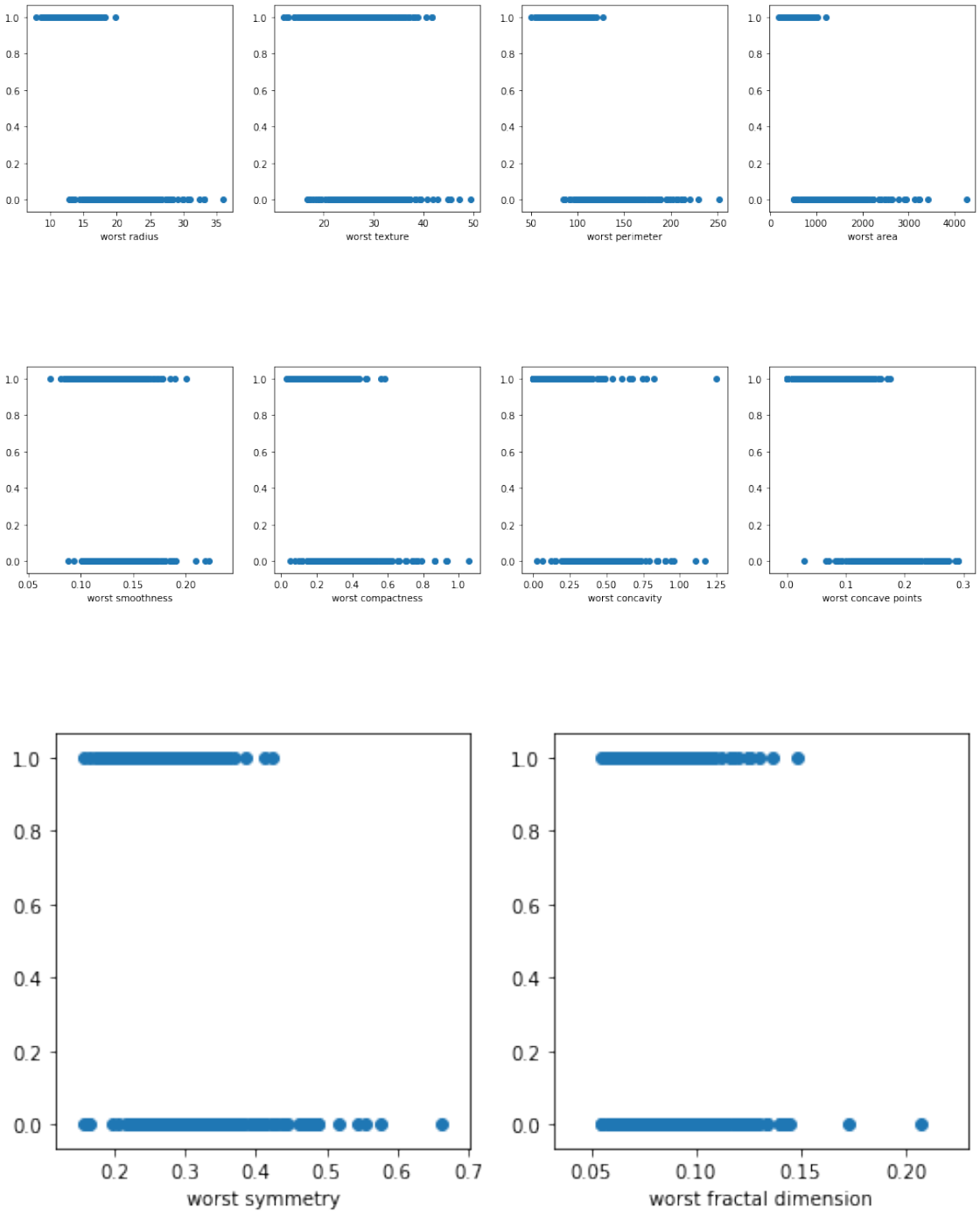
```
df=pd.DataFrame(dat)
df.columns=names # pour ajouter les noms de colonnes
Counter(Y) # nb of cases / controls
df.groupby(Y).mean()
```

[3]:
```
   mean radius  mean texture  mean perimeter   mean area  mean smoothness  \
0    17.462830     21.604906      115.365377  978.376415         0.102898
1    12.146524     17.914762       78.075406  462.790196         0.092478

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0          0.145188        0.160775             0.087990       0.192909
1          0.080085        0.046058             0.025717       0.174186

   mean fractal dimension  ...  worst radius  worst texture  worst perimeter  \
0                0.062680  ...     21.134811      29.318208       141.370330
1                0.062867  ...     13.379801      23.515070        87.005938

    worst area  worst smoothness  worst compactness  worst concavity  \
0  1422.286321          0.144845           0.374824         0.450606
1   558.899440          0.124959           0.182673         0.166238

   worst concave points  worst symmetry  worst fractal dimension
0              0.182237        0.323468                 0.091530
1              0.074444        0.270246                 0.079442

[2 rows x 30 columns]
```

[4]:
```
#-- Plots
for i in range(0,dat.shape[1],4):
    plt.figure(figsize=(18,4))
    for j in range(0,4):
        if i+j < dat.shape[1]:
            plt.subplot(1,4,j+1)
            plt.scatter(dat[:,i+j],Y)
            plt.xlabel(names[i+j])
    plt.show()
```




mean smoothness
mean compactness
mean concavity
mean concave points
mean symmetry
mean fractal dimension
radius error
texture error
perimeter error
area error
smoothness error
compactness error
concavity error
concave points error
symmetry error
fractal dimension error



1.3 Algo de descente du gradient pour la régression logistique

code des fonctions et tests de ces fonctions dans le cas de la régression logistique simple et multiple

[5]:
```
k= 2
k1 = 2
```

```
k2 = 3
X = np.hstack((np.ones((dat.shape[0],1)),dat))
X1 = np.hstack((np.reshape(X[:,0],(dat.shape[0],1)), np.reshape(X[:,k],(dat.
 ↪shape[0],1))))
X2 = np.hstack((np.reshape(X[:,0],(dat.shape[0],1)), np.reshape(X[:,k1],(dat.
 ↪shape[0],1)), np.reshape(X[:,k2],(dat.shape[0],1))))
print(X1)
print(X2)
```

```
[[ 1.   10.38]
 [ 1.   17.77]
 [ 1.   21.25]
 ...
 [ 1.   28.08]
 [ 1.   29.33]
 [ 1.   24.54]]
[[  1.    10.38 122.8 ]
 [  1.    17.77 132.9 ]
 [  1.    21.25 130.  ]
 ...
 [  1.    28.08 108.3 ]
 [  1.    29.33 140.1 ]
 [  1.    24.54  47.92]]
```

[6]:
```
###### -- standardisation des X
def standardisation(x):
    x_data = x[:,1:]
    return np.hstack((x[:,:1],(x_data - x_data.mean(axis=0)) / x_data.
 ↪std(axis=0,ddof=0)))

#-- Test 1 - reg simple
X1_standard = standardisation(X1)
print(stats.describe(X1_standard))
print(X1_standard)

#-- Test 2 - reg multiple
X2_standard = standardisation(X2)
print(X2_standard)
```

```
DescribeResult(nobs=569, minmax=(array([ 1.        , -2.22924851]), array([1.
, 4.65188898])), mean=array([1.00000000e+00, 1.04973636e-16]),
variance=array([0.        , 1.00176056]), skewness=array([0.        ,
0.64873357]), kurtosis=array([-3.        ,  0.74114542]))
[[ 1.         -2.07333501]
 [ 1.         -0.35363241]
 [ 1.          0.45618695]
 ...
```

```
 [ 1.          2.0455738 ]
 [ 1.          2.33645719]
 [ 1.          1.22179204]]
[[ 1.         -2.07333501  1.26993369]
 [ 1.         -0.35363241  1.68595471]
 [ 1.          0.45618695  1.56650313]
 ...
 [ 1.          2.0455738   0.67267578]
 [ 1.          2.33645719  1.98252415]
 [ 1.          1.22179204 -1.81438851]]
```

[7]:
```
#-- Modèle
# attention X : contient X^0 = [1,...,1] + les X^j
def f(x,B):
    exb = np.exp(np.dot(x,B))
    return np.divide(exb,1+exb)

#-- Test 1 - reg simple
B1 = [2,3]
print(X1)
print(f(X1_standard,[0,0.2]))

#-- Test 2 - reg multiple
B2 = [2,3,4]
#print(f(X2_standard,B2))
```

```
[[ 1.   10.38]
 [ 1.   17.77]
 [ 1.   21.25]
 ...
 [ 1.   28.08]
 [ 1.   29.33]
 [ 1.   24.54]]
[0.39779359 0.48232575 0.52279354 0.51268388 0.44266252 0.45833011
 0.50803178 0.51791483 0.52940751 0.55504789 0.54583463 0.48372038
 0.56376568 0.55401316 0.5385567  0.59483326 0.50977653 0.51617156
 0.53323209 0.44289218 0.45844566 0.42097021 0.44174413 0.54352615
 0.52430278 0.46642856 0.52604365 0.51117215 0.56913763 0.4508297
 0.56731064 0.49313967 0.55435813 0.58277686 0.48360415 0.52673983
 0.52824784 0.48988273 0.56833855 0.51779863 0.5266238  0.52395453
 0.56388013 0.51152103 0.52929156 0.4803505  0.47152835 0.49267435
 0.44599492 0.53496927 0.52685584 0.46573365 0.48778942 0.49313967
 0.53172588 0.49372134 0.49162743 0.52673983 0.5002368  0.41361434
 0.44887159 0.51965767 0.53323209 0.43715828 0.55274787 0.55389817
 0.52000618 0.49709529 0.47721462 0.46747116 0.52349016 0.44610992
 0.56055815 0.4593702  0.46781876 0.50419271 0.40405344 0.463534
 0.55424314 0.48499901 0.51954149 0.4601794  0.56502433 0.5807385
```

0.45775244 0.49104584 0.52523132 0.56136052 0.5291756 0.45301997
0.55481799 0.54029115 0.44749035 0.48848714 0.50628684 0.54341067
0.48313925 0.50756648 0.42551463 0.50558882 0.56582486 0.43224047
0.51431164 0.50128397 0.50000409 0.45671292 0.48883602 0.49127847
0.5044254 0.52290965 0.47326833 0.51710136 0.50419271 0.51047436
0.45983257 0.52604365 0.45902346 0.46955721 0.54202463 0.50838075
0.40270934 0.47477684 0.51059066 0.40349322 0.46631273 0.47582146
0.5625062 0.49558278 0.46631273 0.56742488 0.43064215 0.50221479
0.52615969 0.43796005 0.53045094 0.53693704 0.46955721 0.45475042
0.48000199 0.43178366 0.41564764 0.48558027 0.47698241 0.46087317
0.44990804 0.44622492 0.46851405 0.49395401 0.45232811 0.48395284
0.51733379 0.51640402 0.45417348 0.42824738 0.45417348 0.48430155
0.51686892 0.50198209 0.42437731 0.42688045 0.51035806 0.46110446
0.48674292 0.53404287 0.53195764 0.50547248 0.39034799 0.49430302
0.56239166 0.47280427 0.42040309 0.50396002 0.41474359 0.44691507
0.45198224 0.44392594 0.48569653 0.50954391 0.53404287 0.42801949
0.52998722 0.58390807 0.5118699 0.44933218 0.53624266 0.45152116
0.49174375 0.47558931 0.47791134 0.4606419 0.54444977 0.524651
0.48755685 0.58718353 0.54548848 0.46561784 0.53485348 0.5296394
0.53716847 0.51082326 0.50314559 0.50035315 0.58503839 0.55171211
0.4919764 0.46967314 0.476518   0.51128845 0.53774697 0.42631122
0.53311624 0.49593181 0.49046427 0.57244423 0.55240267 0.47257226
0.48906862 0.47895657 0.52639175 0.64872312 0.42915928 0.43761639
0.47953733 0.51117215 0.47361641 0.43269739 0.45555835 0.45613558
0.55424314 0.5353166 0.49756069 0.58966324 0.66279991 0.59785814
0.46075753 0.52279354 0.58842395 0.52523132 0.59830565 0.71715193
0.45717489 0.45071447 0.48720801 0.55194232 0.54883269 0.50663584
0.47837586 0.44002323 0.5685669 0.44944735 0.54952406 0.49023165
0.5061705 0.47431263 0.5004695 0.47396451 0.60855167 0.476518
0.54537309 0.66019455 0.589438   0.54375708 0.5330004 0.50105127
0.53230526 0.63427489 0.49604816 0.5296394 0.46422848 0.51280016
0.4712964 0.42779162 0.51977384 0.4579835 0.56009951 0.47756297
0.4405967 0.50803178 0.48313925 0.45232811 0.58424726 0.43899137
0.48709174 0.49395401 0.45833011 0.48965012 0.51710136 0.42870327
0.50779913 0.49535009 0.50512346 0.49779339 0.46202979 0.47872428
0.43624237 0.43041393 0.41994954 0.48662664 0.48697546 0.54410345
0.49546643 0.50698484 0.55297797 0.49209272 0.48685919 0.56021418
0.45971697 0.44335157 0.42403628 0.43692926 0.49790974 0.45821457
0.4315553 0.40159037 0.4919764 0.47164433 0.43967922 0.4951174
0.49546643 0.47338435 0.46388122 0.50430906 0.43098452 0.52581157
0.45267402 0.4768663 0.42596978 0.48418531 0.51652025 0.53010316
0.45613558 0.50070221 0.50663584 0.4477205 0.49686259 0.51989001
0.44139983 0.52488313 0.47953733 0.55769   0.46805051 0.47152835
0.44979286 0.52778389 0.45544291 0.44691507 0.49546643 0.44726022
0.46214547 0.44979286 0.47419659 0.49918962 0.47872428 0.57449337
0.43956456 0.49744434 0.49186007 0.46422848 0.45590467 0.48871973
0.48581279 0.52650778 0.49476838 0.48837085 0.47280427 0.52894368
0.58684504 0.48523151 0.47628581 0.53033501 0.54641144 0.42972947

```
0.45140591 0.47744685 0.46260825 0.46260825 0.51082326 0.60232573
0.45198224 0.49465204 0.42688045 0.44944735 0.53982873 0.47895657
0.43555572 0.54641144 0.43979388 0.46364973 0.45602012 0.54548848
0.41847643 0.47152835 0.50791546 0.53473769 0.48174473 0.47547323
0.49535009 0.47872428 0.44829598 0.4764019  0.52012235 0.40371728
0.48837085 0.46376547 0.44967768 0.49186007 0.44864133 0.5241867
0.5159391  0.48406907 0.48000199 0.47141238 0.52778389 0.5327687
0.62001957 0.52186457 0.52801587 0.5208193  0.41791024 0.524651
0.49709529 0.43853294 0.46202979 0.49814244 0.49802609 0.52314186
0.45002323 0.53126231 0.46897764 0.48116376 0.53763128 0.48127995
0.50291289 0.53114641 0.47268826 0.50384367 0.50291289 0.46156709
0.50361098 0.45786797 0.47570538 0.57084873 0.4593702  0.48871973
0.47164433 0.56479555 0.60031738 0.48104758 0.50058586 0.51431164
0.52615969 0.56605352 0.60254867 0.43853294 0.47512502 0.62994571
0.61430148 0.56890936 0.56753912 0.60221425 0.59045129 0.58051185
0.58864936 0.48918491 0.48755685 0.50977653 0.51686892 0.48616158
0.54687281 0.48709174 0.4906969  0.60154511 0.44944735 0.62177238
0.45740589 0.45867677 0.5144279  0.46897764 0.44553498 0.50256384
0.48534776 0.49942232 0.43944991 0.48081521 0.4078695  0.46654439
0.47164433 0.4945357  0.46376547 0.51059066 0.53658987 0.42995759
0.51477666 0.42540086 0.51454416 0.51070696 0.48697546 0.47698241
0.47942118 0.52267743 0.47036875 0.56021418 0.46550204 0.50628684
0.42585598 0.42893126 0.5087297  0.47477684 0.45833011 0.55401316
0.446685   0.44680003 0.51431164 0.43486933 0.49744434 0.49209272
0.51500916 0.51128845 0.48755685 0.46990499 0.43761639 0.52685584
0.50628684 0.49290701 0.45809903 0.42847531 0.49372134 0.41904284
0.42927331 0.43235469 0.47988582 0.50849707 0.46561784 0.51605533
0.48058285 0.51826344 0.53786266 0.55964077 0.57164667 0.57107675
0.44381105 0.56593919 0.57084873 0.60065234 0.51640402 0.54571925
0.46584946 0.46851405 0.50058586 0.55700106 0.52546343 0.53658987
0.61584414 0.53079869 0.61021344 0.59561819 0.50349463 0.59864117
0.53936623 0.55378316 0.59045129 0.61518328 0.62886011 0.56708214
0.53601117 0.60277157 0.6008756  0.61474248 0.56078743]
```

[9]:
```python
#-- Fonction-cout
# ordre des arguments : beta en 1er pour comparer avec la fonction d'optim
def cout(B,x,y):
    fB = f(x,B)
    return -1/x.shape[0] * (np.dot(y,np.log(fB)) + np.dot(1-y,np.log(1-fB)))

#%% Test 1 - reg simple
print(cout([1,1],X1_standard,Y))
#%% Test 2 - reg multiple
print(cout(B2,X2_standard,Y))
```

```
0.9738532957118856
4.208726878326237
```

[54]:
```
#-- Gradient
# ordre des arguments : beta en 1er pour comparer avec la fonction d'optim
def grad(B,x,y,lamb=0):
    return 1/x.shape[0] * np.dot((f(x,B)-y),x) + np.dot(B,lamb/x.shape[0])

#%% Test 1 - reg simple
print(grad([1,1],X1_standard,Y))

#%% Test 2 - reg multiple
print(grad(B2,X2_standard,Y))
```

```
[0.0666003  0.36447474]
[-0.06182767  0.49539301  0.65301784]
```

[11]:
```
#-- Algo de descente du gradient
def grad_descent(init,x,y,pas,epsi,ITE_MAX,lamb=0):
    nbIt = 0
    x_normalize = standardisation(x)
    cout_arr = []
    b_arr = []
    B = np.asarray(init).astype(float)
    new_B = B.copy()
    b_arr.append(B.tolist())
    old_cout = 0
    new_cout = cout(B,x_normalize,y)
    cout_arr.append(new_cout)
    while(np.abs(new_cout-old_cout)>epsi and nbIt < ITE_MAX):
        new_B = B - pas * grad(B,x_normalize,y,lamb=lamb)
        B = new_B.copy()
        old_cout = new_cout
        new_cout = cout(B,x_normalize,y)
        b_arr.append(B.tolist())
        cout_arr.append(new_cout)
        nbIt += 1
    return b_arr,cout_arr

#-- Test 1  - reg simple
b, c = grad_descent((0,0.2),X1,Y,0.05,0.01,500)
print("b",b)
print("c",c)
#-- Test 2 - reg multiple
print()
b, c = grad_descent((0,0,0),X2,Y,0.05,0.01,500)
print(b)
print(c)
```

```
b [[0.0, 0.2], [0.006375985168497839, 0.18749318218632108]]
c [0.7382645091114638, 0.7343470742956743]
```

```
[[0.0, 0.0, 0.0], [0.0063708260105448155, -0.010036949633874742,
-0.01795293670311326]]
[0.6931471805599446, 0.6839472611673111]
```

[34]:
```
from scipy.special import expit

def displayCout(data_c,title):
    plt.figure(1,figsize=(10,6))
    plt.grid()
    plt.title("Evolution du cout "+title)
    x = np.linspace(0, len(data_c), len(data_c))
    plt.plot(x, data_c)
    plt.xlabel("Nb iterations")
    plt.ylabel("Cout")
    plt.show()

def display(title,data_X,data_Y,data_b,data_c=None):
    print("\nBetas :",data_b[-1])
    plt.figure(1,figsize=(10,6))
    plt.title(title)
    plt.scatter(standardisation(data_X)[:,1],data_Y)
    x_lim = plt.xlim()
    x = np.linspace(x_lim[0],x_lim[1],100)
    for i in range(0,len(data_b),10):
        y = expit(np.dot(x,data_b[i][1])+data_b[i][0])
        if i == 0 :
            plt.plot(x,y,color='green')
        elif i + 10 >= len(data_b):
            plt.plot(x,y,color='red')
        else:
            plt.plot(x,y,color='gray')
    plt.xlabel(names[k-1])
    plt.ylabel("Breast Cancer")
    plt.show()
    if data_c != None:
        displayCout(data_c,"regression logistique simple")

def display3D(title,data_X,data_Y,data_b,data_c=None):
    print("\nBetas :",data_b[-1])
    fig = plt.figure(1,figsize=(12,8))
    ax = fig.add_subplot(111, projection='3d')
    ax.set_title(title)
    xx = standardisation(data_X)
    ax.scatter(xx[:,1],xx[:,2],data_Y,c='r', marker='o')
    x_lim = plt.xlim()
    y_lim = plt.ylim()
```

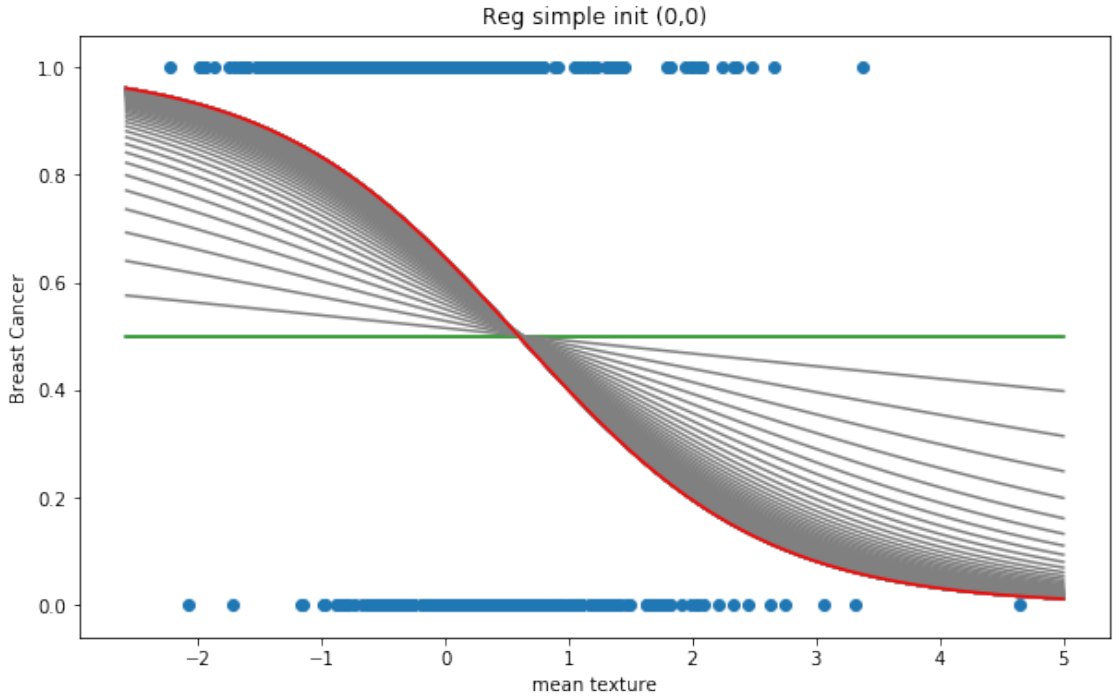
```
    x = np.arange(x_lim[0], x_lim[1], 0.05)
    y = np.arange(y_lim[0], y_lim[1], 0.05)
    x, y = np.meshgrid(x, y)
    z = expit(np.dot(data_b[-1][2],x)+np.dot(data_b[-1][1],y)+data_b[-1][0])
    z = z.reshape(x.shape)
    ax.plot_wireframe(x,y,z,color='gray')
    ax.set_xlabel(names[k1-1])
    ax.set_ylabel(names[k2-1])
    ax.set_zlabel("Breast Cancer")
    plt.show()
    if data_c != None:
        displayCout(data_c,"regression lineaire multiple")
```

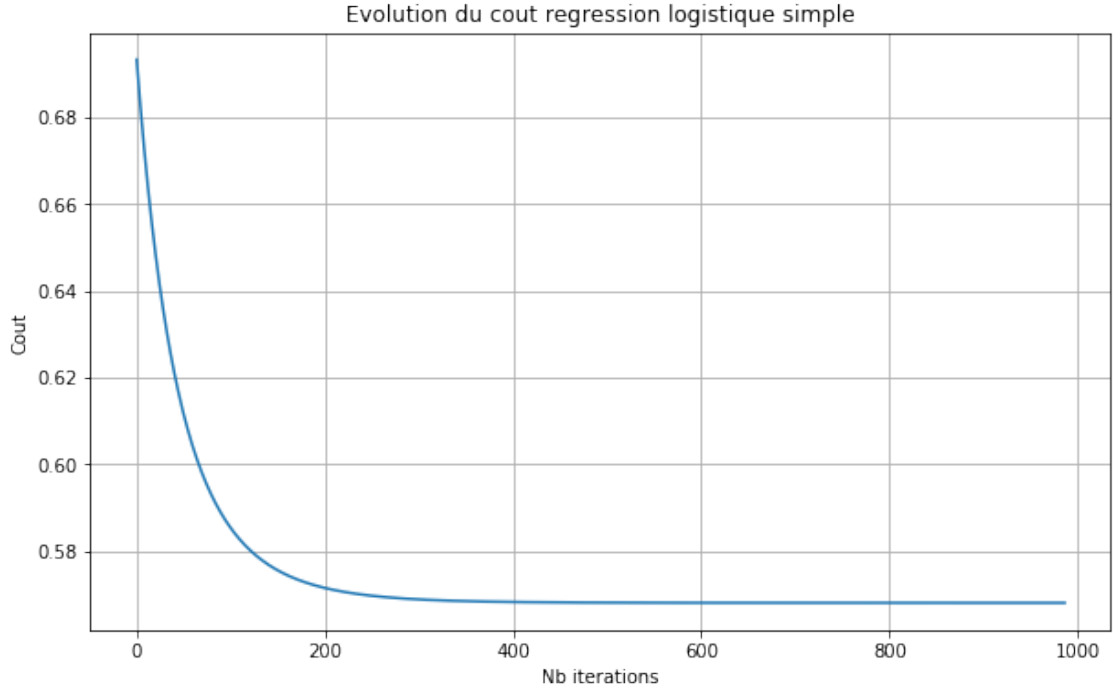
1.4 Test avec plusieurs initialisations

visualisation de la fonction cout et des valeurs des paramètres au cours des itérations

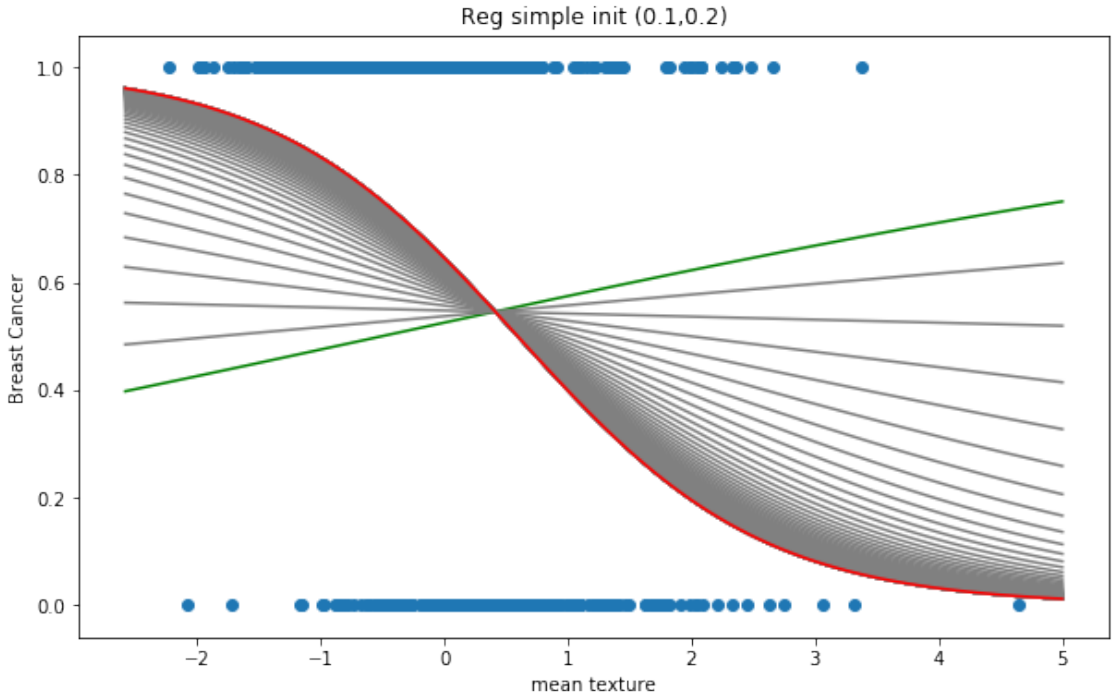
[13]:
```
#-- Test 1  - reg simple

# init 1
b, c = grad_descent((0,0),X1,Y,0.05,1e-9,10000)
display("Reg simple init (0,0)",X2_standard,Y,b,c)
# init 2
b, c = grad_descent((0.1,0.2),X1,Y,0.05,1e-9,10000)
display("Reg simple init (0.1,0.2)",X1,Y,b,c)
# init 3
b, c = grad_descent((1,2),X1,Y,0.05,1e-9,10000)
display("Reg simple init (1,2)",X1,Y,b,c)


#-- Test 2 - reg multiple

# init 1
b, c = grad_descent((0,0,0),X2,Y,0.05,1e-9,10000)
display3D("Reg simple multiple (0,0,0)",X2,Y,b,c)
# init 2
b, c = grad_descent((0.,0.2,0.3),X2,Y,0.05,1e-9,10000)
display3D("Reg simple multiple (0.1,0.2,0.3)",X2,Y,b,c)
# init 3
b, c = grad_descent((1,2,3),X2,Y,0.05,1e-9,10000)
display3D("Reg simple multiple (1,2,3)",X2,Y,b,c)
# ...
```

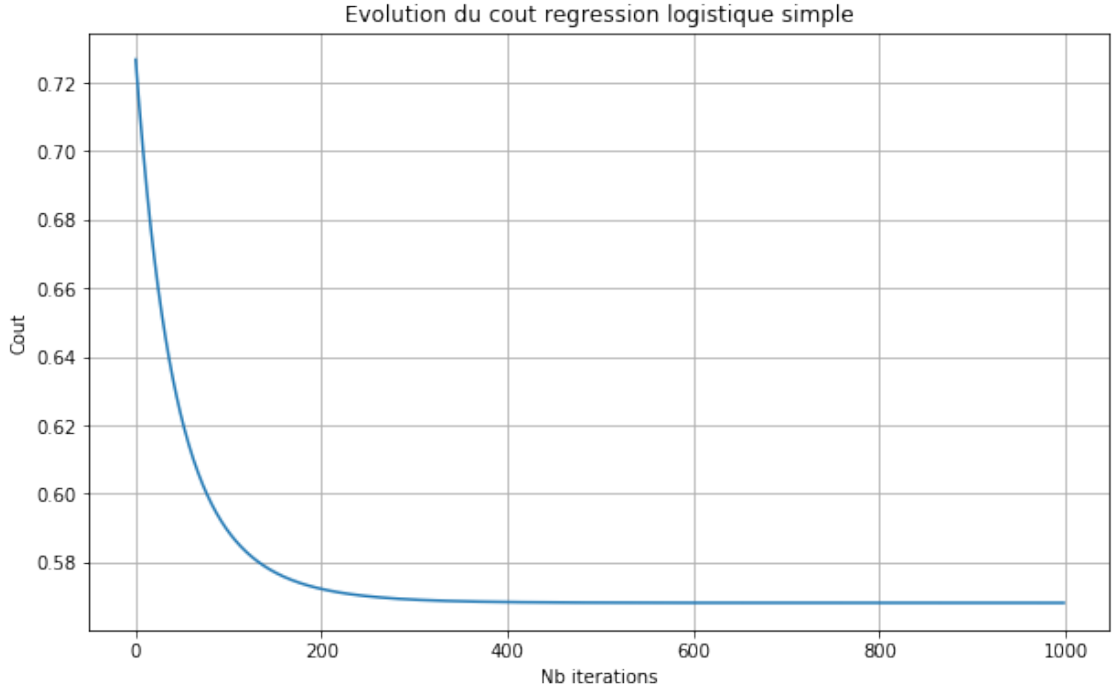
```
Betas : [0.5992537239557103, -1.00733204188107]
```



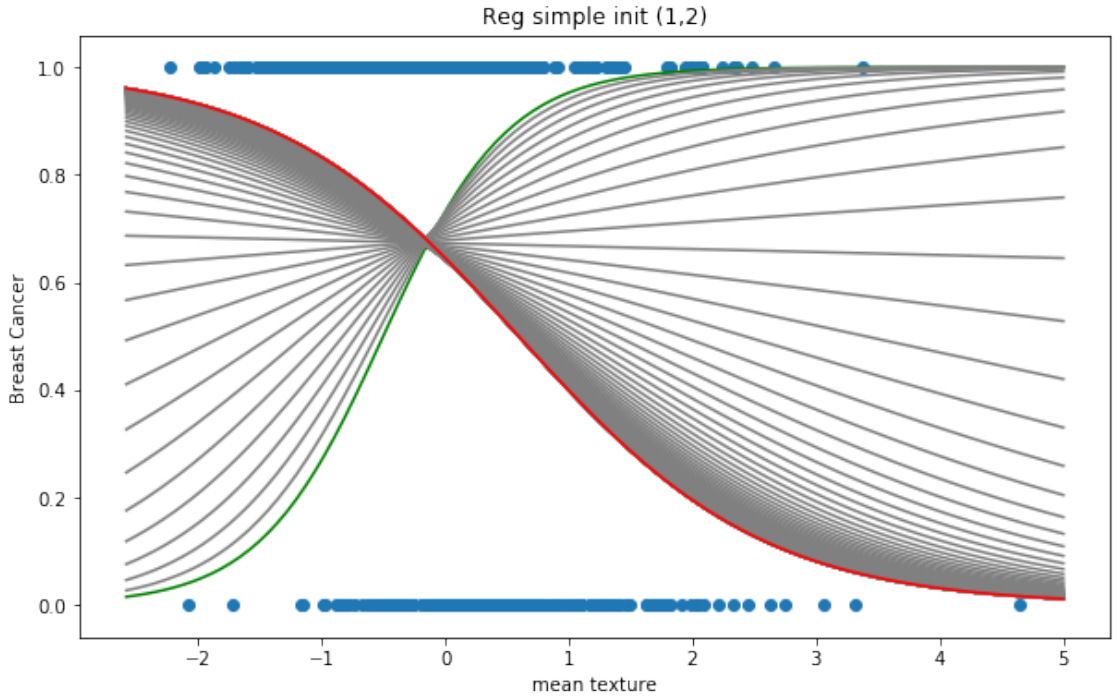


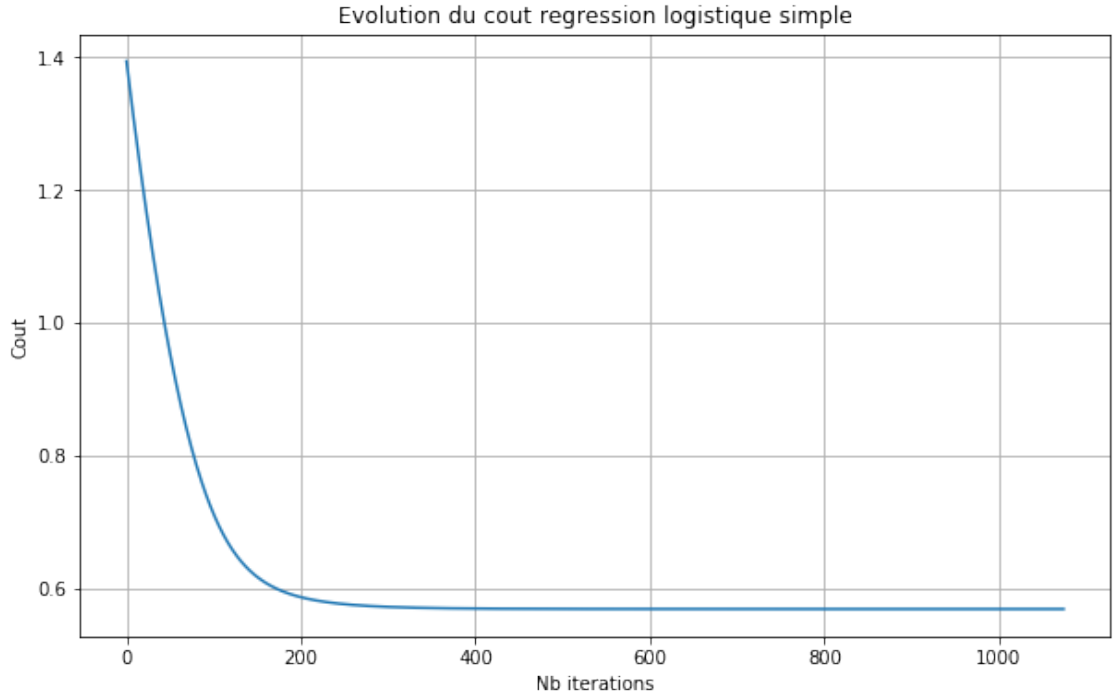

```
Betas : [0.5992591490200806, -1.0073313109521806]
```



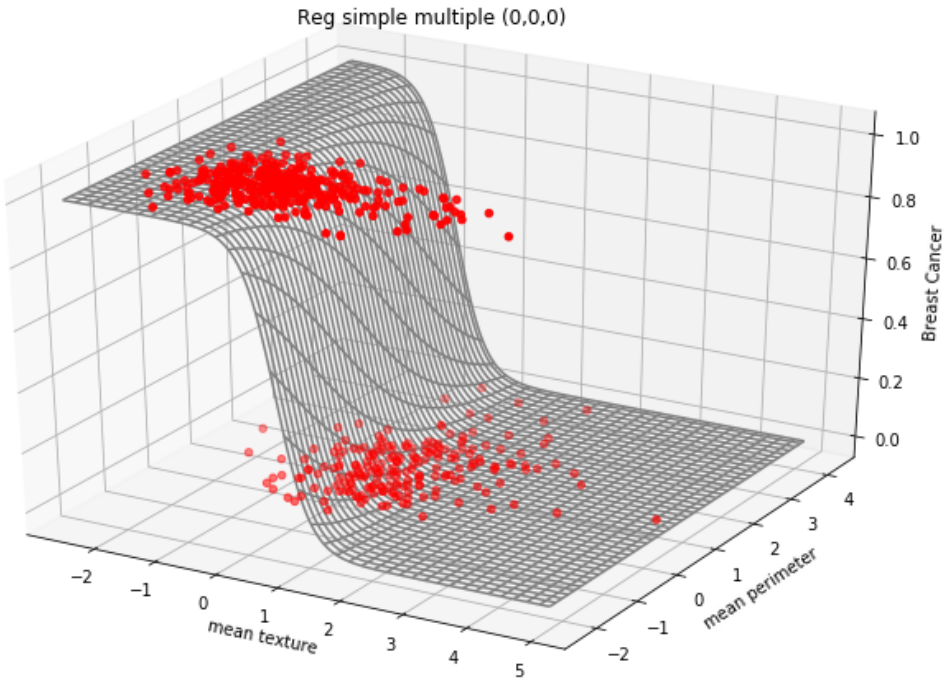


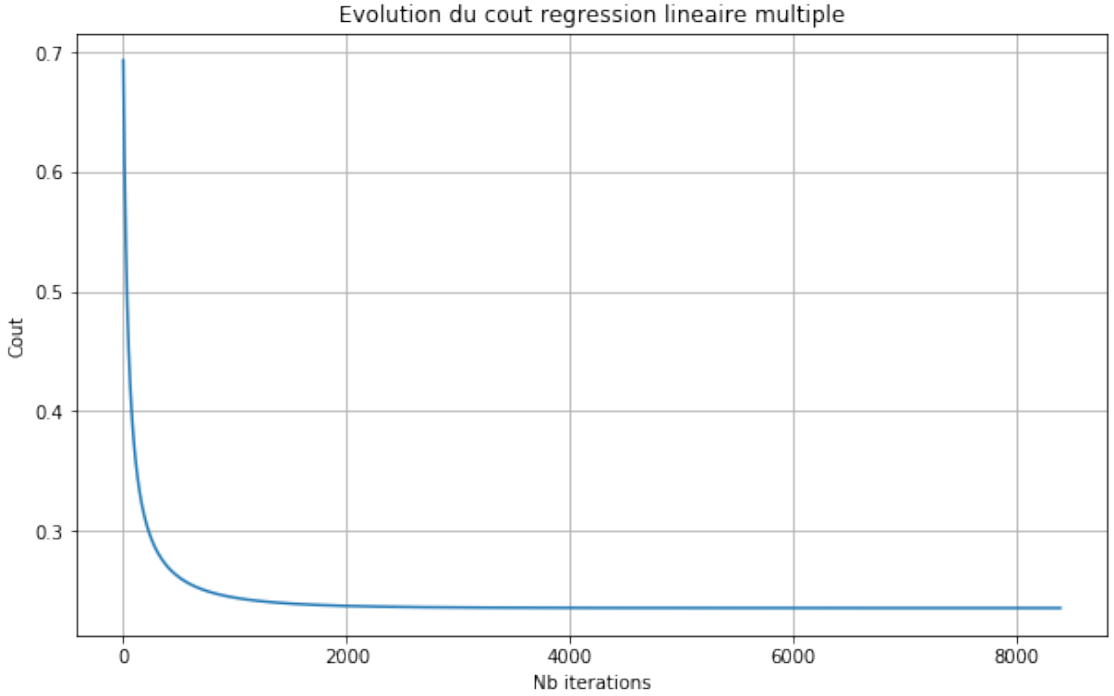

```
Betas : [0.5992802919220761, -1.0073227990860303]
```



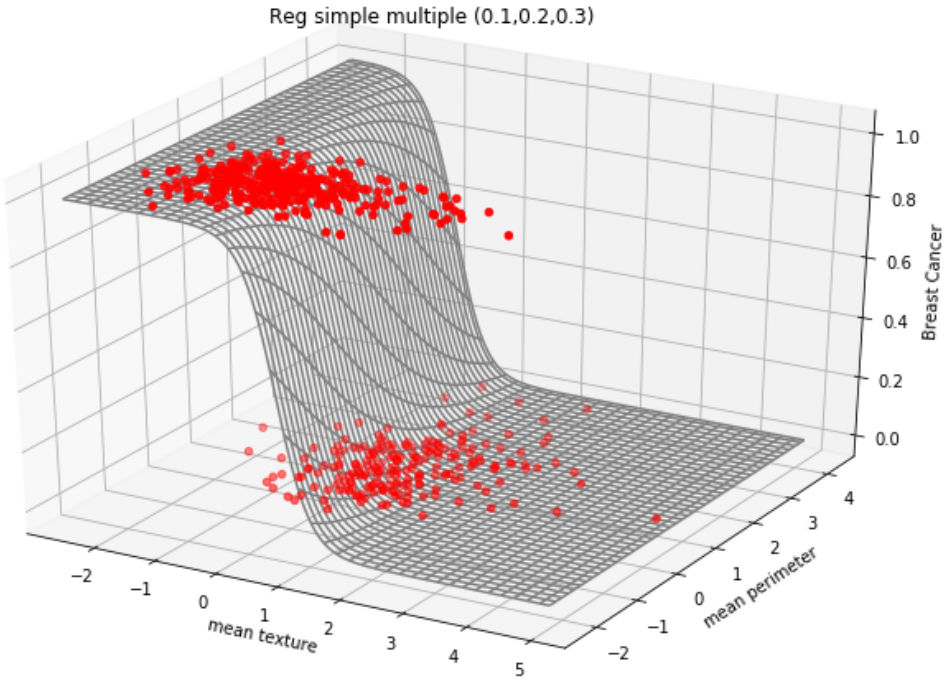


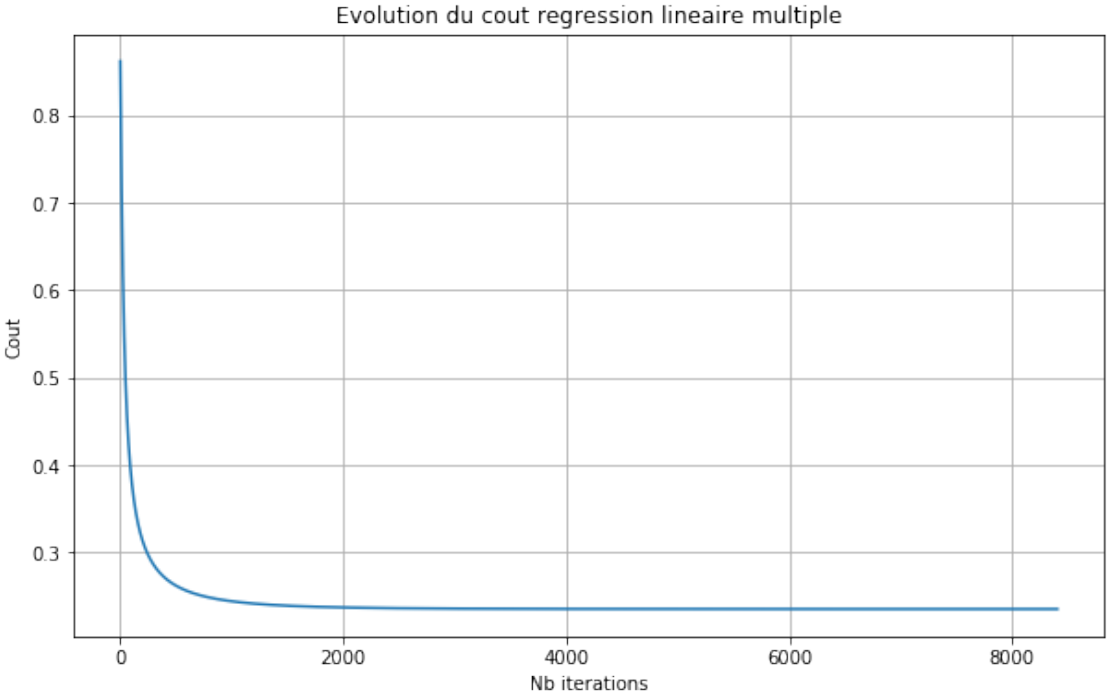
Betas : [0.6989425278488138, -0.9541363368593714, -4.0598881070996224]


Reg simple multiple (0,0,0)
mean texture
mean perimeter
Breast Cancer
1.0
0.8
0.6
0.4
0.2
0.0
−2
−1
0
1
2
3
4
5

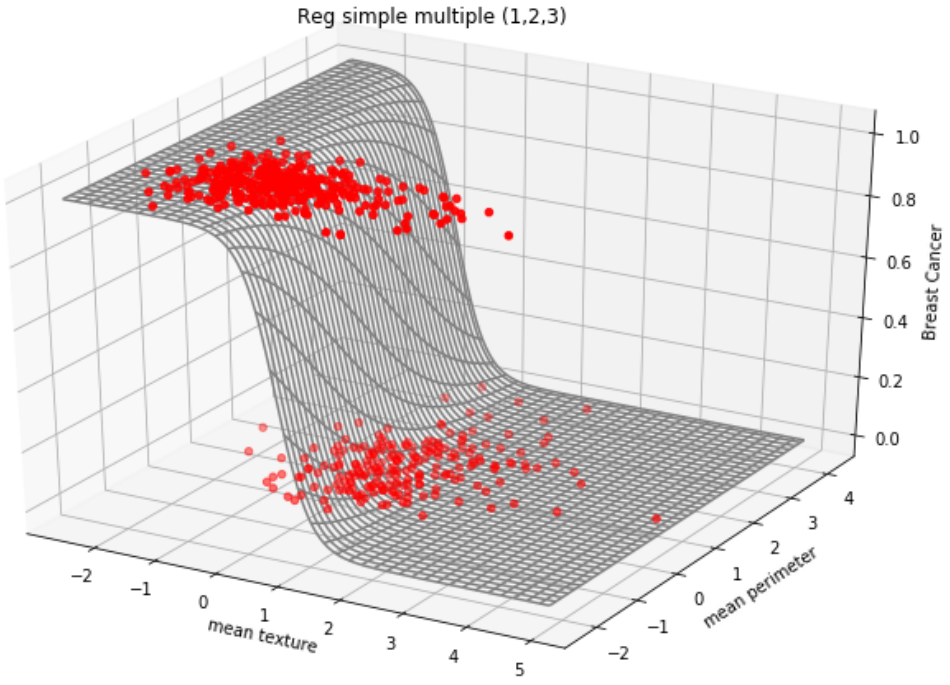

Evolution du cout regression lineaire multiple
Cout
Nb iterations
0.7
0.6
0.5
0.4
0.3
0
2000
4000
6000
8000

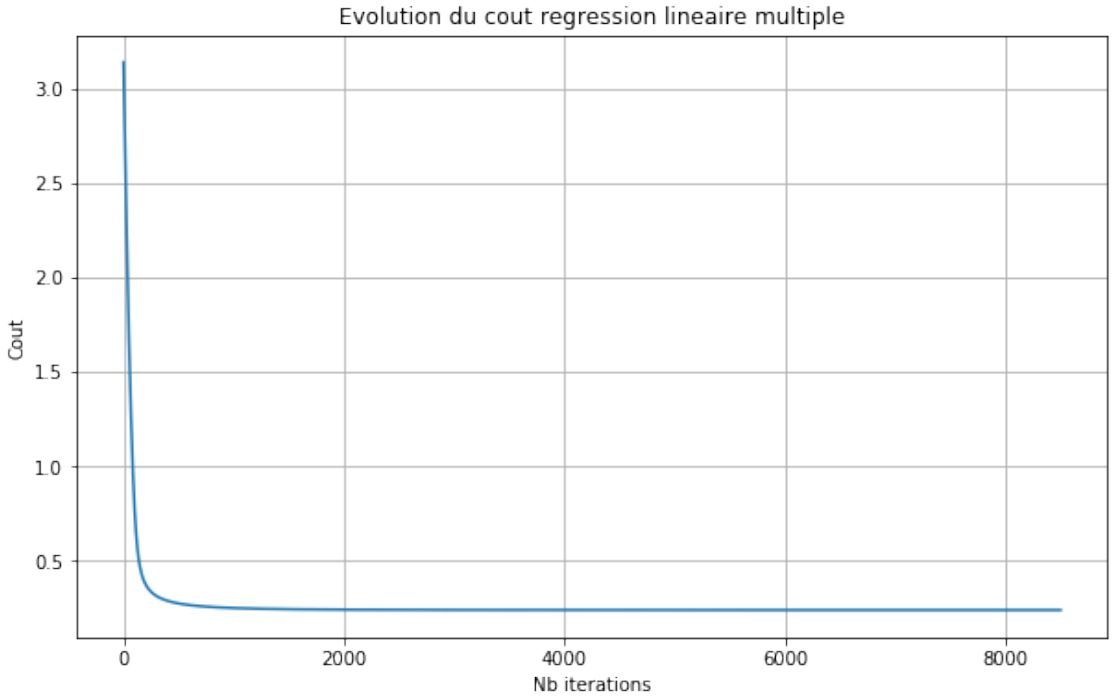
```
Betas : [0.6989424621807573, -0.9541358629868639, -4.059884574690401]
```





```
Betas : [0.6989425308767911, -0.9541363587083905, -4.059888269967913]
```





COMMENTAIRES:

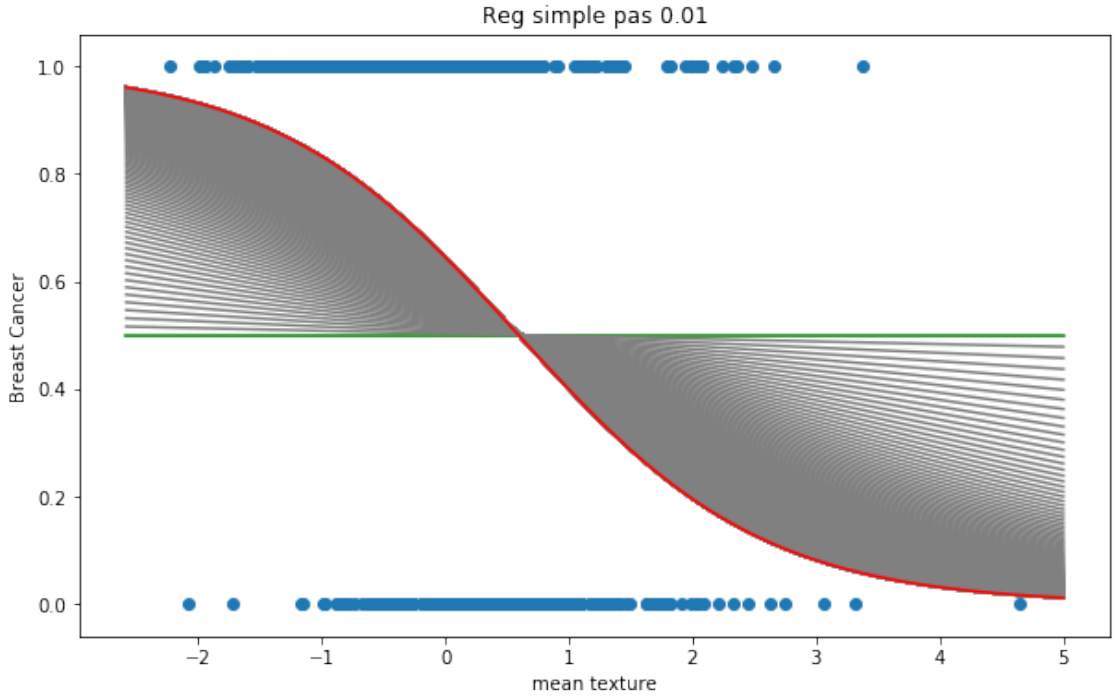
1.5 Test avec plusieurs pas de descente

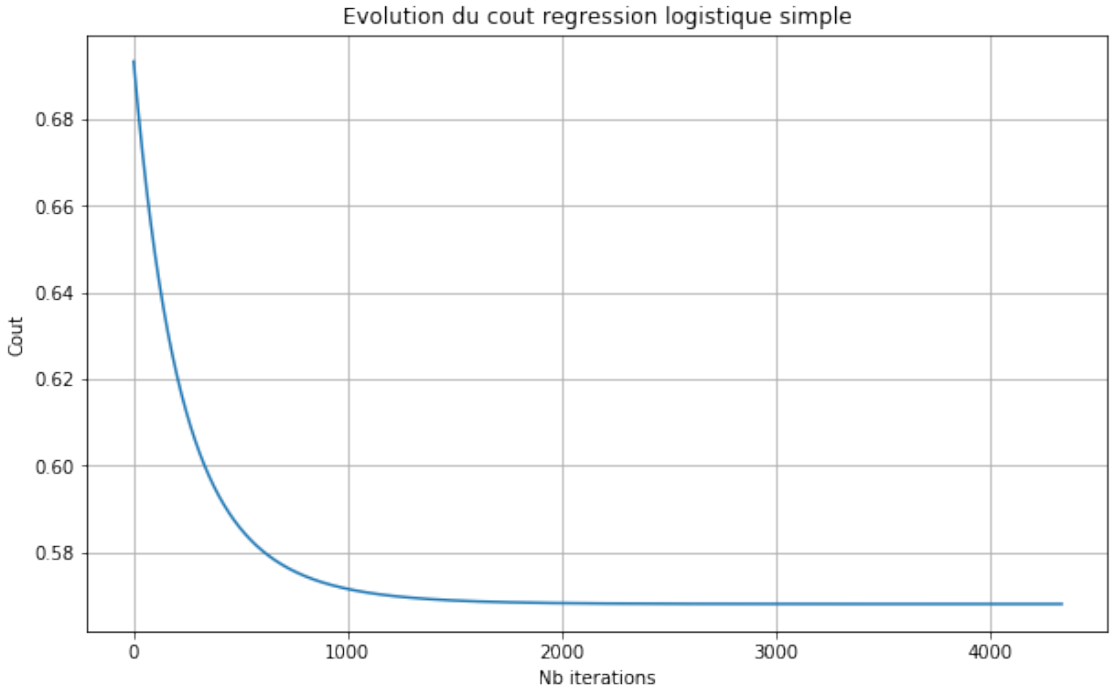
visualisation de la fonction cout et des valeurs des paramètres au cours des itérations

[14]:
```
#-- Test 1  - reg simple

# alpha 1
b, c = grad_descent((0,0),X1,Y,0.01,1e-9,10000)
display("Reg simple pas 0.01",X1,Y,b,c)
# alpha 2
b, c = grad_descent((0,0),X1,Y,0.1,1e-9,10000)
display("Reg simple pas 0.1",X1,Y,b,c)
# alpha 3
b, c = grad_descent((0,0),X1,Y,0.001,1e-9,10000)
display("Reg simple pas 0.001",X1,Y,b,c)
# ...



#-- Test 2 - reg multiple

# alpha 1
# init 1
b, c = grad_descent((0,0,0),X2,Y,0.01,1e-9,10000)
```

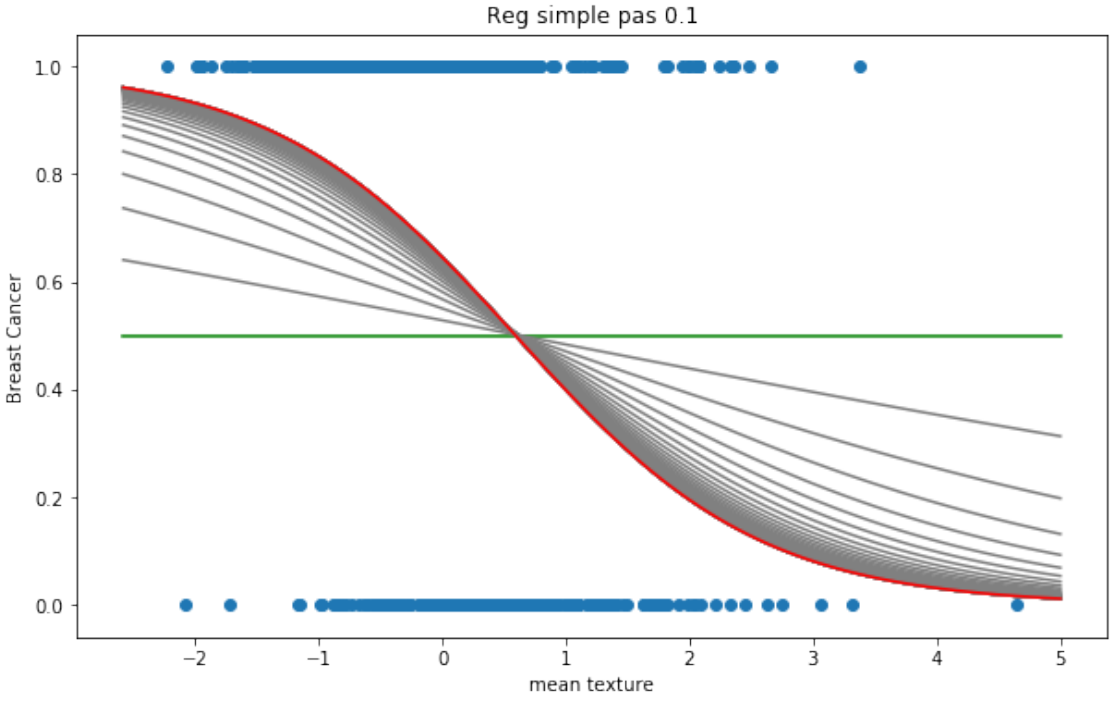
```
display3D("Reg simple multiple pas 0.01",X2,Y,b,c)
# alpha 2
# init 1
b, c = grad_descent((0,0,0),X2,Y,0.1,1e-9,10000)
display3D("Reg simple multiple pas 0.1",X2,Y,b,c)
# alpha 3
# init 1
b, c = grad_descent((0,0,0),X2,Y,0.001,1e-9,10000)
display3D("Reg simple multiple pas 0.001",X2,Y,b,c)
# ...
```

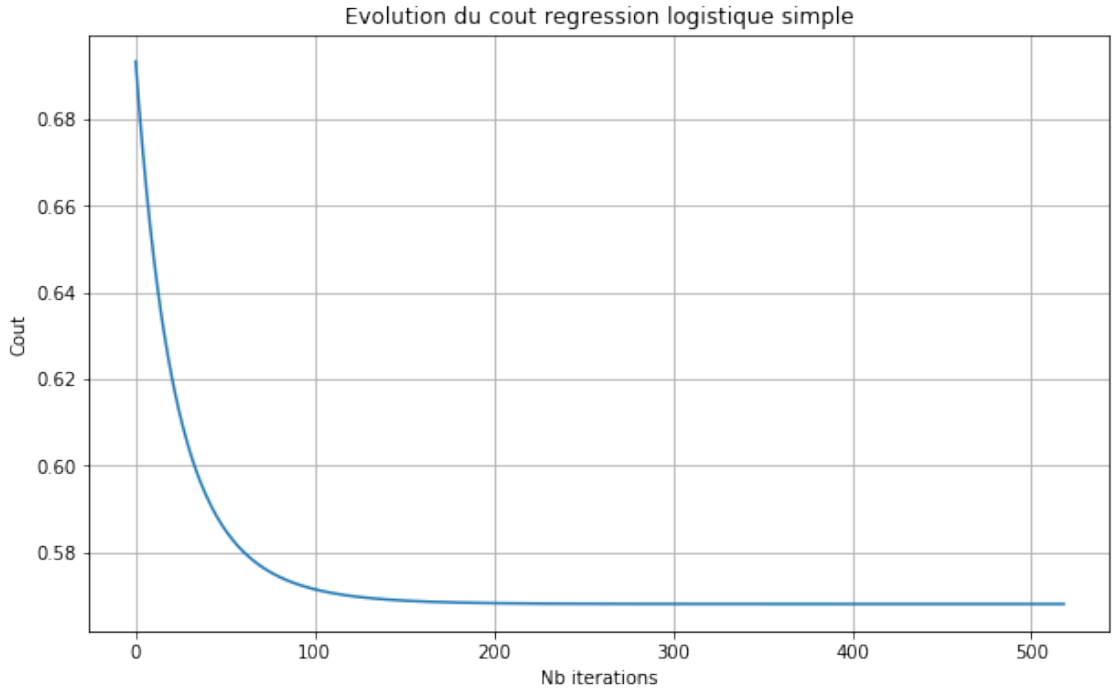
```
Betas : [0.598764216749767, -1.0061103311541422]
```



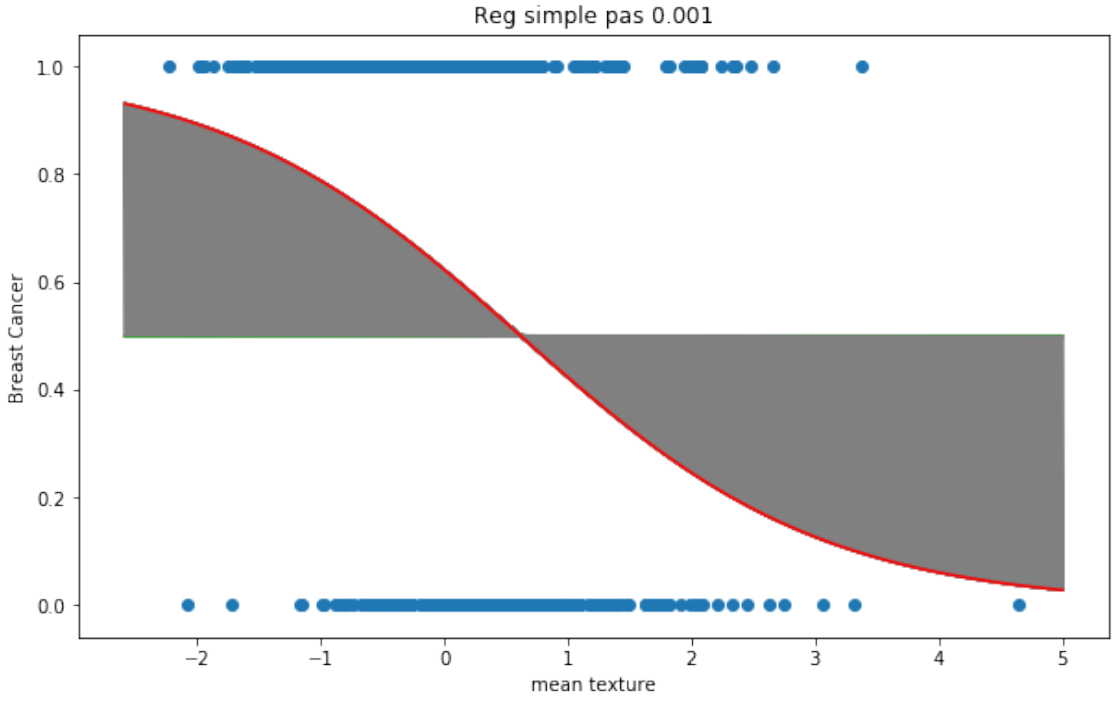


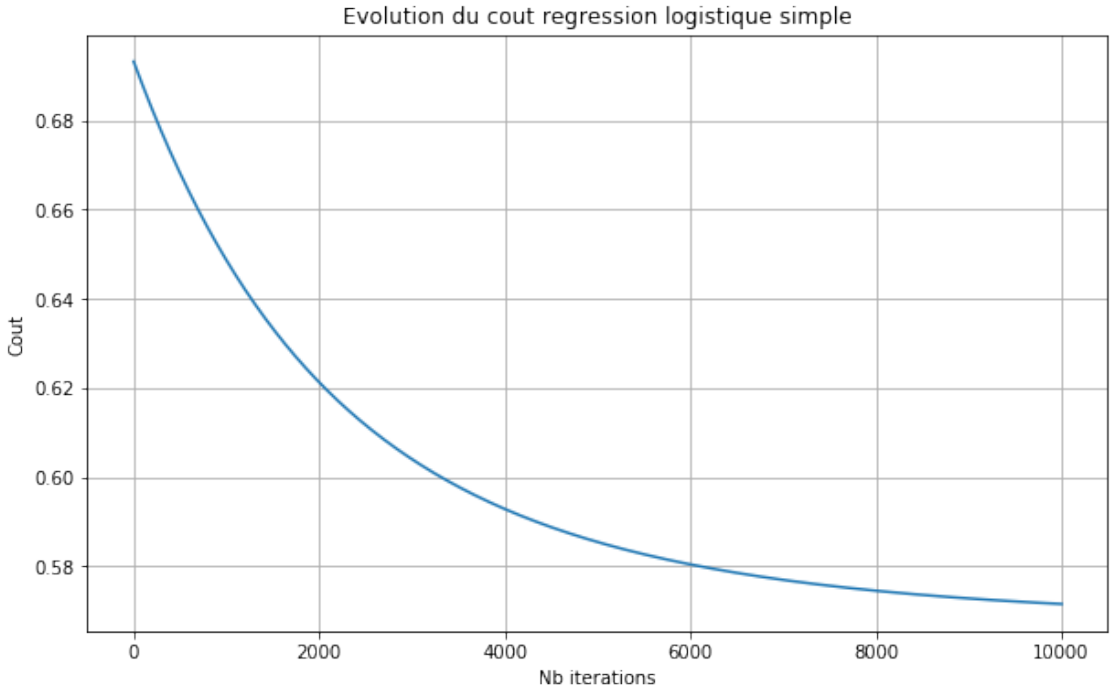

```
Betas : [0.5993690590887206, -1.0076230354805487]
```



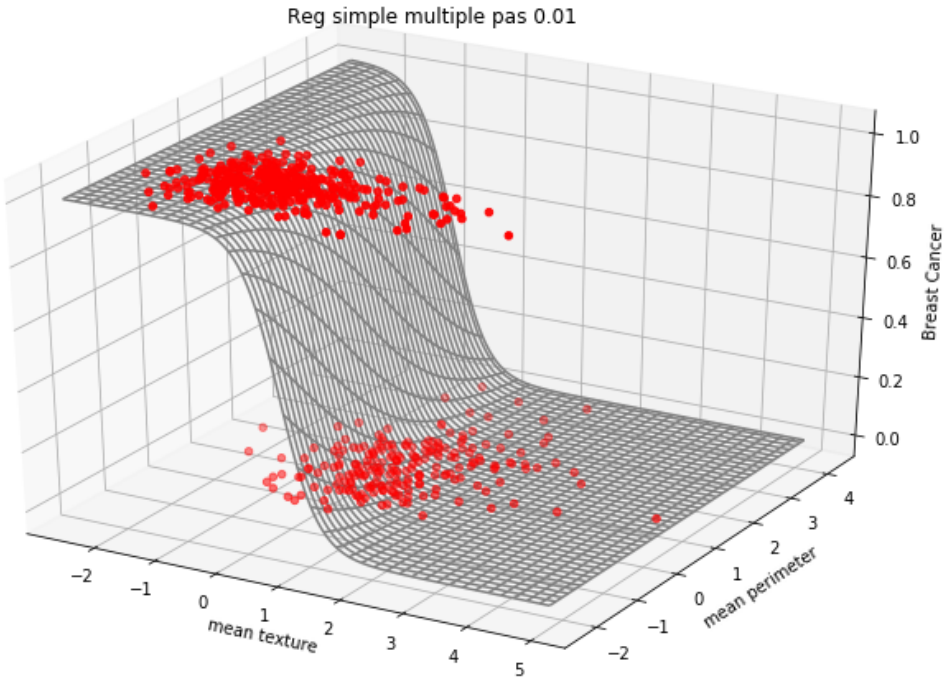


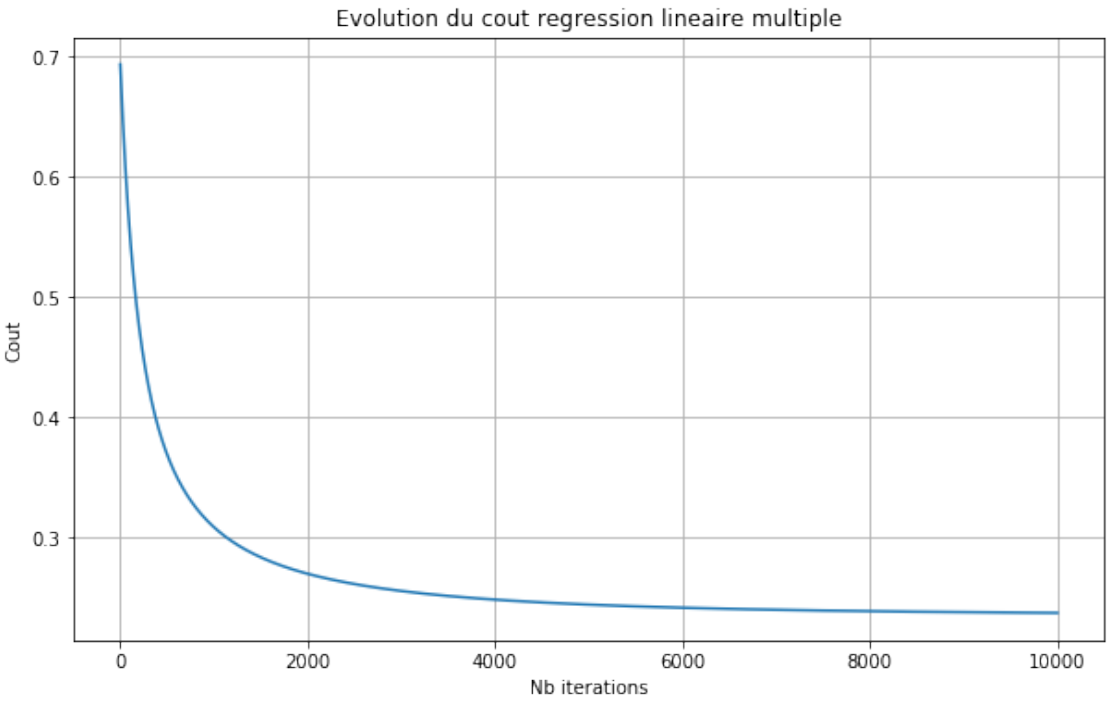

```
Betas : [0.5005575666338329, -0.8111876631169949]
```



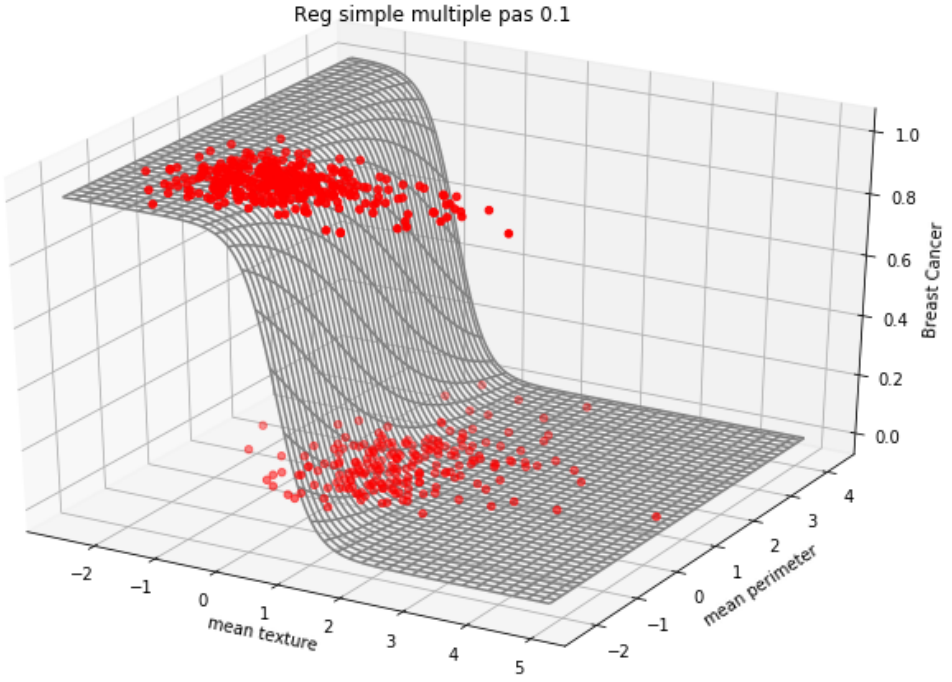


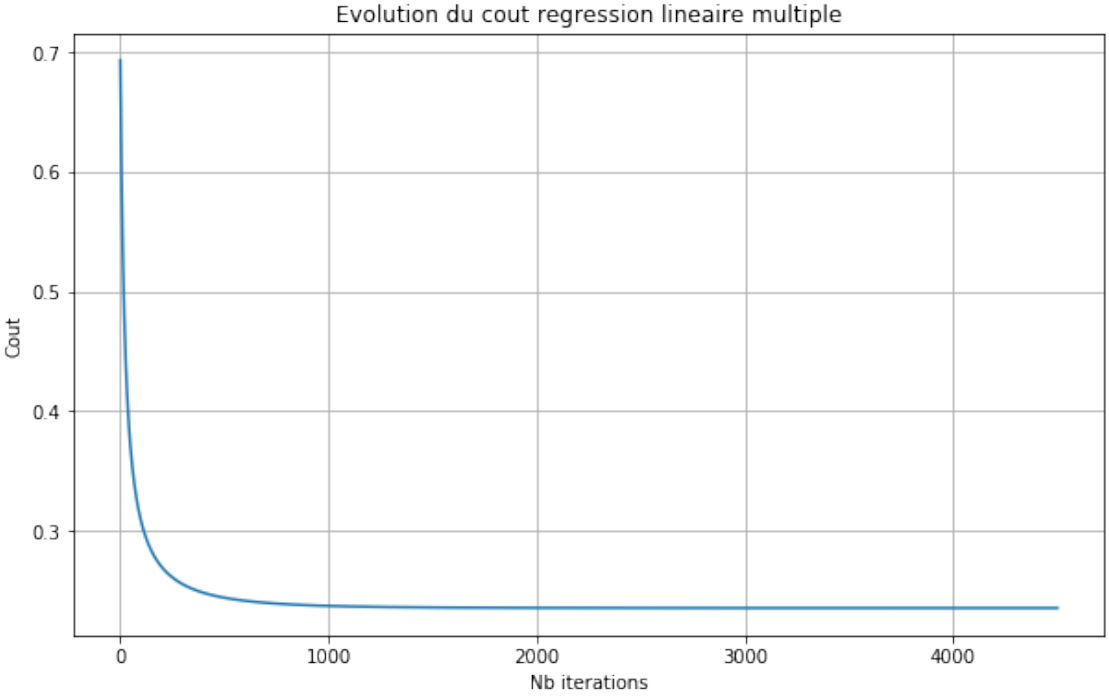
Betas : [0.6905152452121541, -0.8877601122655197, -3.545827201531127]



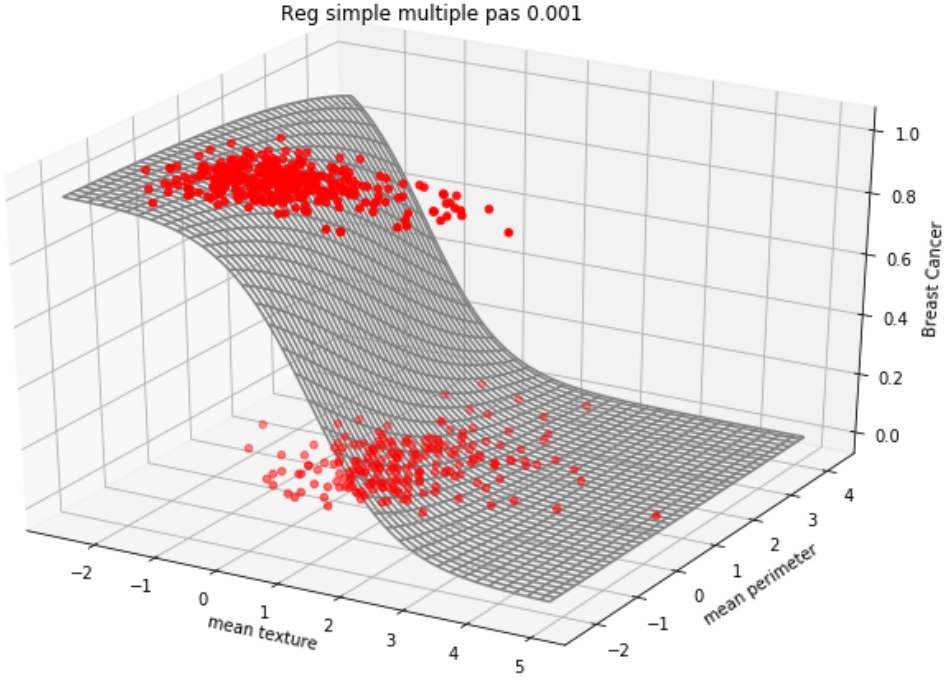


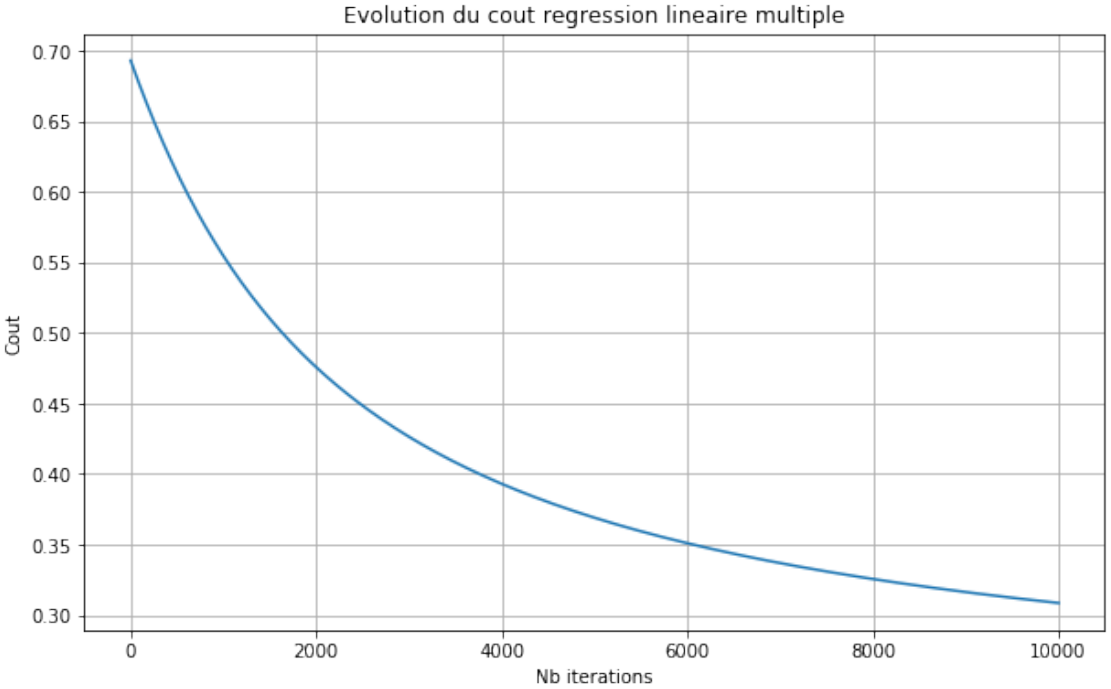

```
Betas : [0.6990100419013534, -0.9546231429615133, -4.063516069080992]
```


Reg simple multiple pas 0.1
Breast Cancer
mean texture
mean perimeter


Evolution du cout regression lineaire multiple
Cout
Nb iterations

```
Betas : [0.5155098222900298, -0.6096412343092106, -1.552996892159287]
```

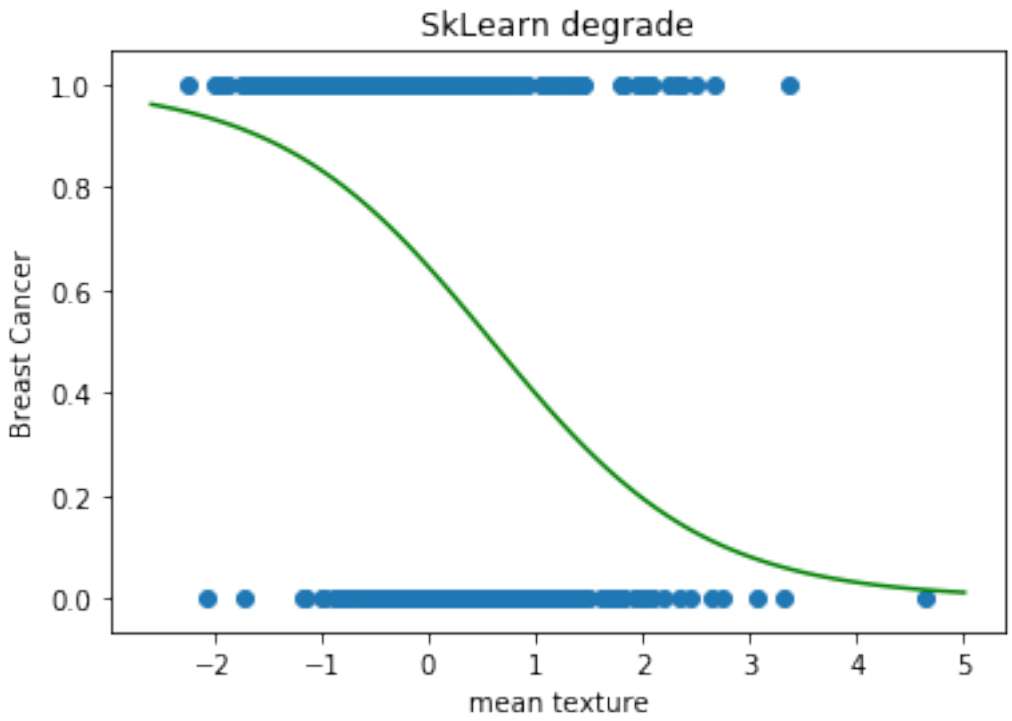




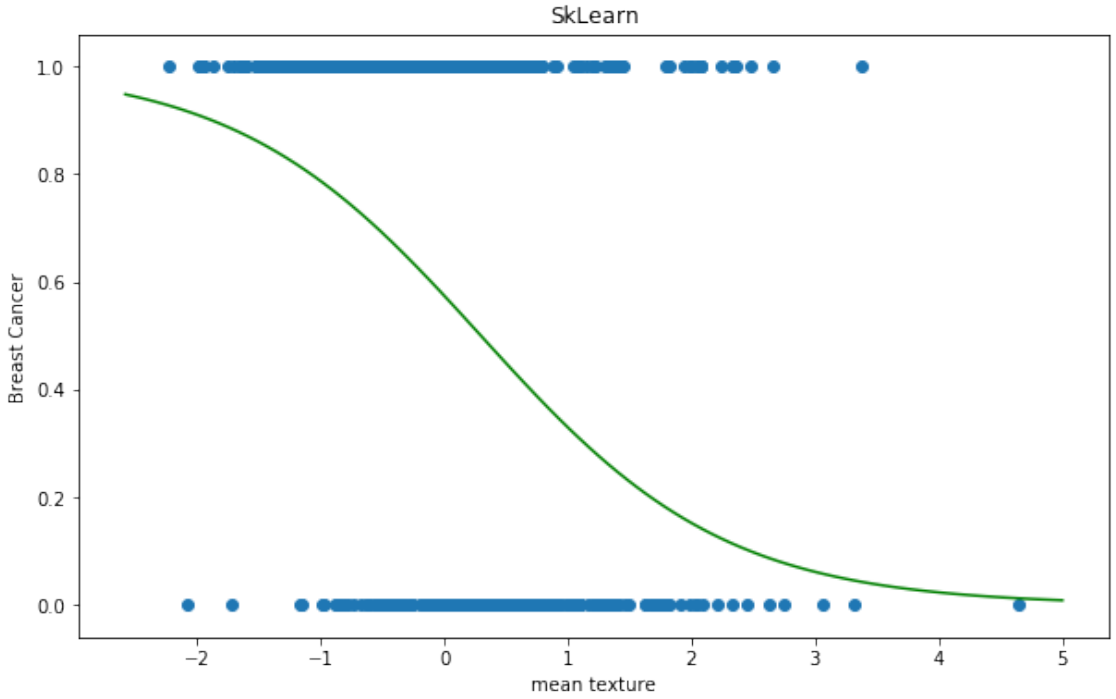
COMMENTAIRES:

[35]:
```
#-- Pour comparer
clf_degrade = LogisticRegression(C=1000,max_iter = 10000,fit_intercept =␣
 ↪False,solver='lbfgs')
clf_degrade.fit(X1_standard, Y)

clf = LogisticRegression(C=1e5, solver='lbfgs')
clf.fit(X1_standard, Y)

clf3D = LogisticRegression(C=1e5, solver='lbfgs')
clf3D.fit(X2_standard, Y)
# and plot the result
plt.clf()
display("SkLearn degrade",X1,Y,clf_degrade.coef_)
display("SkLearn",X1,Y,clf.coef_)
display3D("SkLearn 3D",X2,Y,clf3D.coef_)
# avec les algos d'optim
opti = optimize.fmin_bfgs(cout, x0 = [0,0], fprime = grad, args = (X1_standard,␣
 ↪Y), gtol = 1e-9)
display("Optim",X1,Y,[opti])
opti3D = optimize.fmin_bfgs(cout, x0 = [0,0,0], fprime = grad, args =␣
 ↪(X2_standard, Y), gtol = 1e-9)
display3D("Optim3D",X2,Y,[opti3D])
```

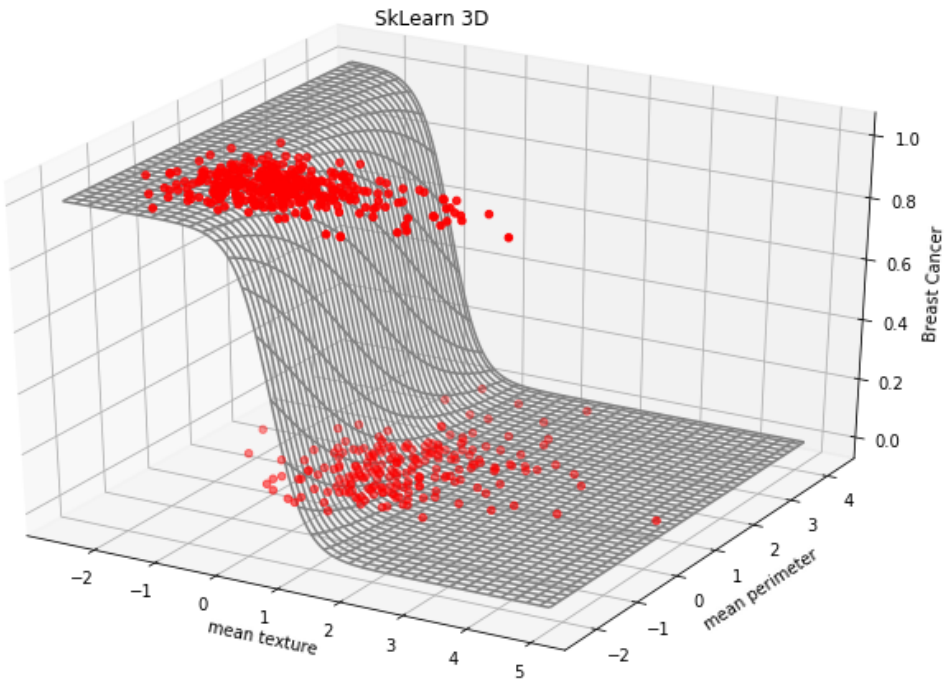
```
Betas : [ 0.59963115 -1.00829703]
```



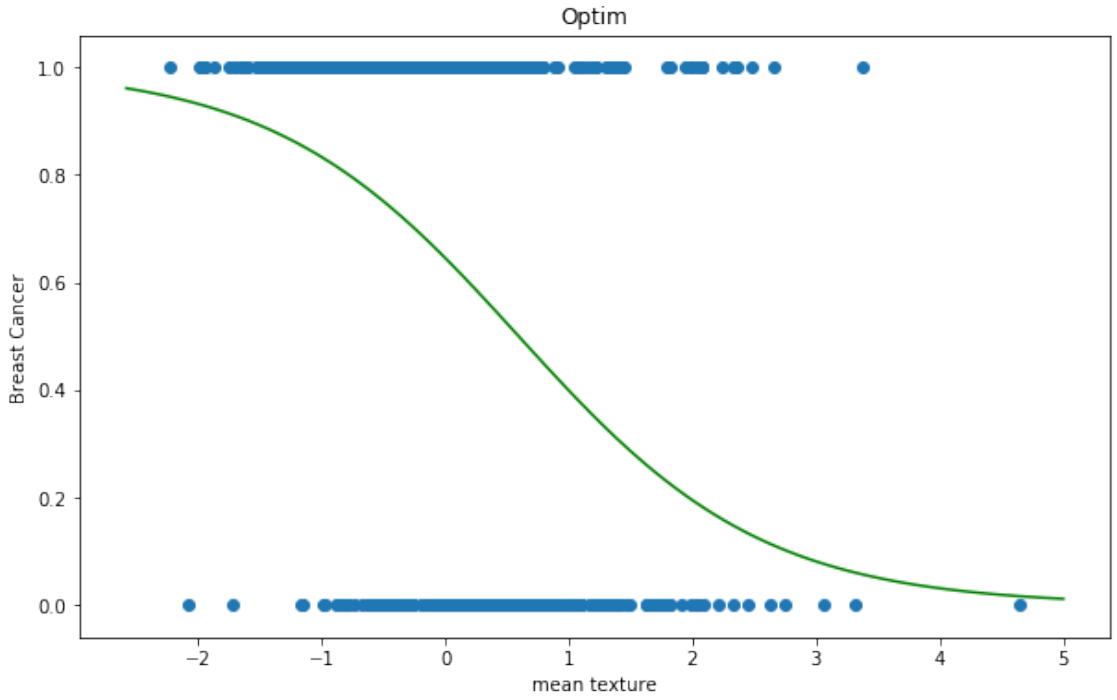

```
Betas : [ 0.29981896 -1.00831064]
```



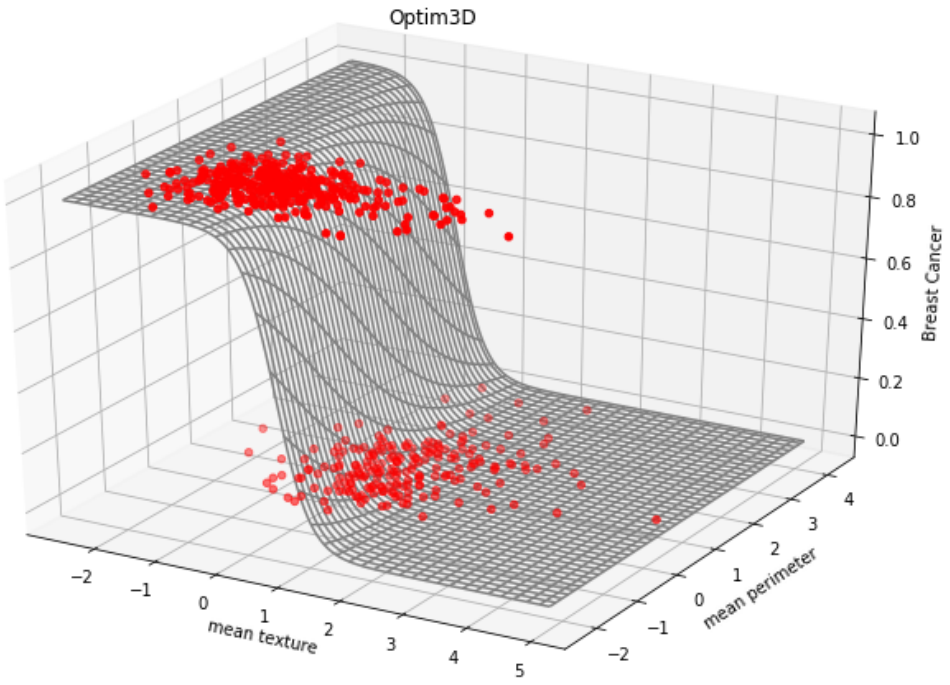
```
Betas : [ 0.34958597 -0.9558018  -4.07229471]
```




```
Optimization terminated successfully.
         Current function value: 0.568119
         Iterations: 12
         Function evaluations: 13
         Gradient evaluations: 13

Betas : [ 0.59963812 -1.00831031]
```




```
Optimization terminated successfully.
         Current function value: 0.235570
         Iterations: 22
         Function evaluations: 23
         Gradient evaluations: 23

Betas : [ 0.6991741  -0.95580287 -4.07230096]
```



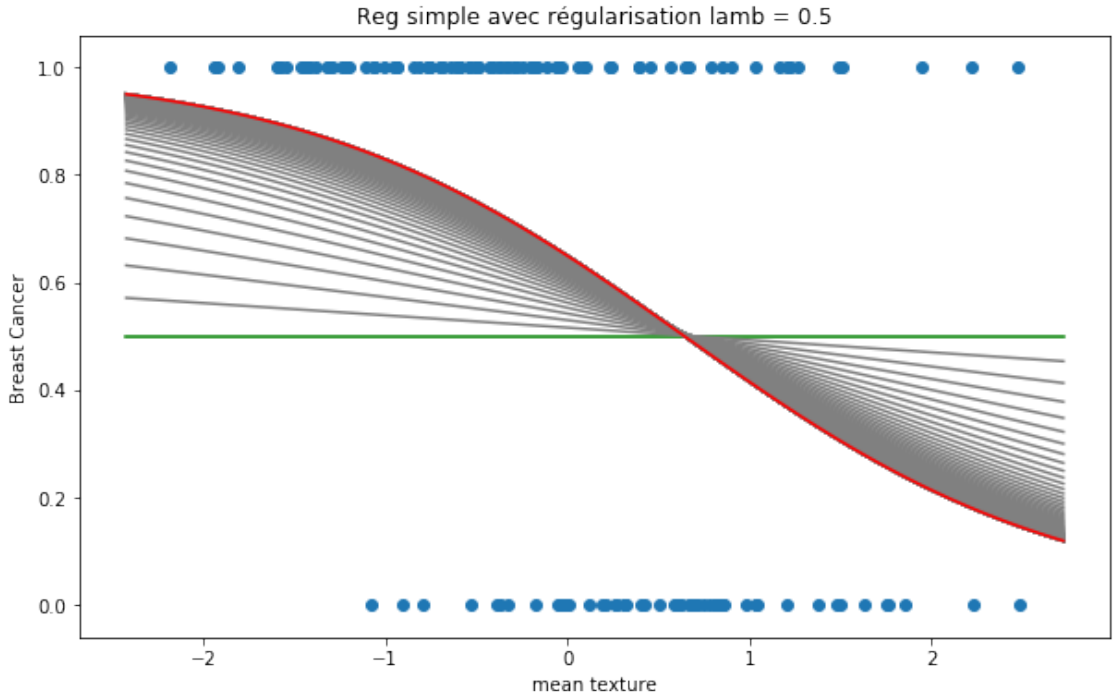
COMMENTAIRES:

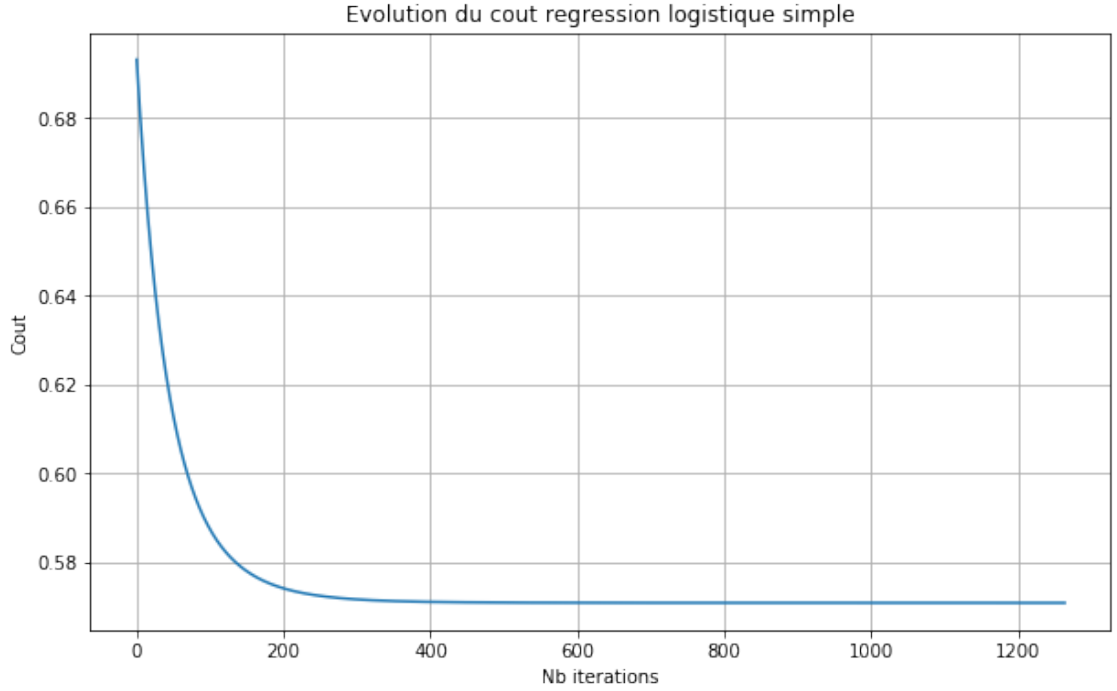
Mes coefficients, ceux de sklearn en mode dégradé et ceux de la fonction optimisée sont les mêmes.

[59]:
```
#Avec la régularisation l2
X_train, X_test, y_train, y_test = train_test_split(X1,Y)
X_train3D, X_test3D, y_train3D, y_test3D = train_test_split(X2,Y)

lamb = 0.5
b, c = grad_descent((0,0),X_train,y_train,0.05,1e-9,10000,lamb=lamb)
display("Reg simple avec régularisation lamb = "+str(lamb),X_test,y_test,b,c)

lamb = 1
b, c = grad_descent((0,0),X_train,y_train,0.05,1e-9,10000,lamb=lamb)
display("Reg simple avec régularisation lamb = "+str(lamb),X_test,y_test,b,c)

lamb = 5
b, c = grad_descent((0,0),X_train,y_train,0.05,1e-9,10000,lamb=lamb)
display("Reg simple avec régularisation lamb = "+str(lamb),X_test,y_test,b,c)

evolutionBetas = []
evolutionErreur = []
lambdas = np.linspace(10e-5,10000,50)
for i in lambdas:
    b, c = grad_descent((0,0),X_train,y_train,0.05,1e-9,10000,lamb=i)
```

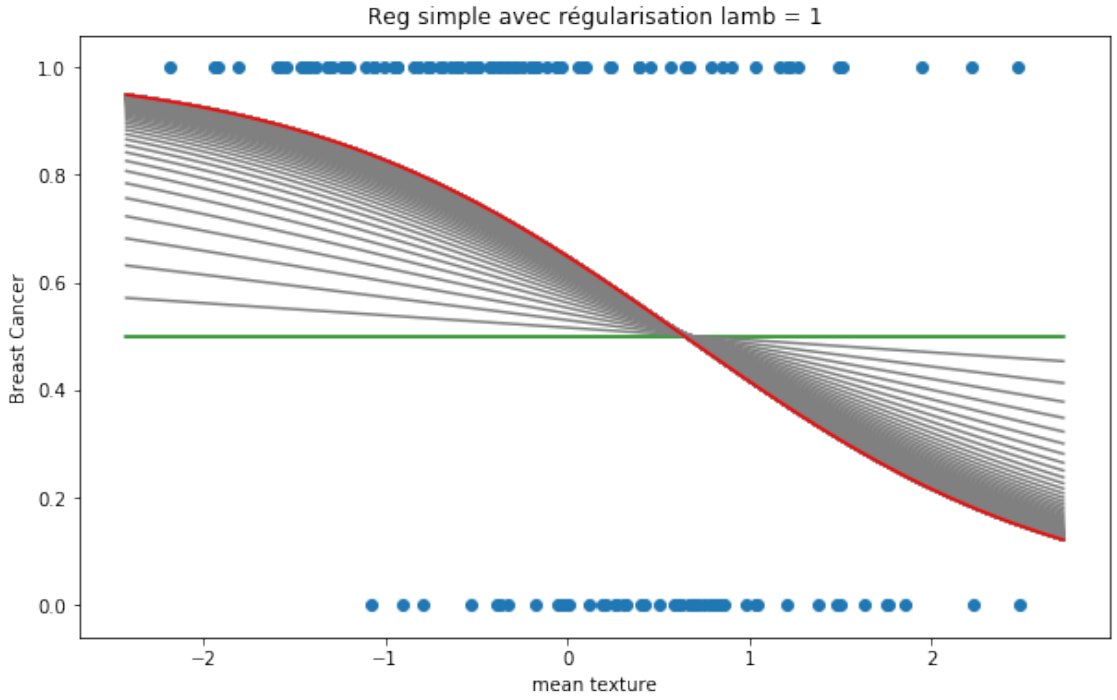
```
    evolutionBetas.append(b[-1])
    evolutionErreur.append(c[-1])

plt.figure()
plt.grid()
plt.title("Évolution des betas en fonction de lambda")
plt.plot(lambdas, evolutionBetas)
plt.show()

plt.figure()
plt.grid()
plt.title("Évolution de l'erreur en fonction de lambda")
plt.plot(lambdas, evolutionErreur)
plt.show()

clf = LogisticRegression(C=1e5, solver='saga',penalty='l1')
clf.fit(standardisation(X_train3D), y_train3D)
display3D("Sklearn penalty l1",X_test3D,y_test3D,clf.coef_)

clf = LogisticRegression(C=1e5, solver='lbfgs',penalty='l2')
clf.fit(standardisation(X_train3D), y_train3D)
display3D("Sklearn penalty l2",X_test3D,y_test3D,clf.coef_)

clf = LogisticRegression(C=1e5, solver='saga',penalty='elasticnet', l1_ratio =␣
↪0.5)
clf.fit(standardisation(X_train3D), y_train3D)
display3D("Sklearn penalty elasticnet",X_test3D,y_test3D,clf.coef_)
```

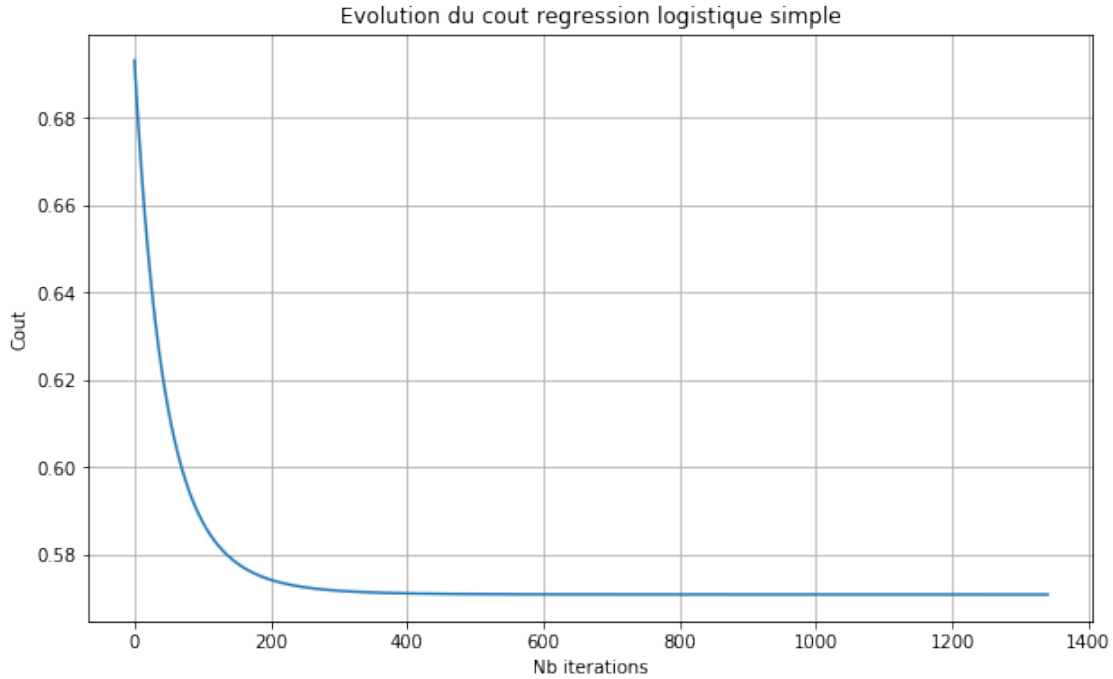
```
Betas : [0.6159110854338111, -0.9583979351202275]
```



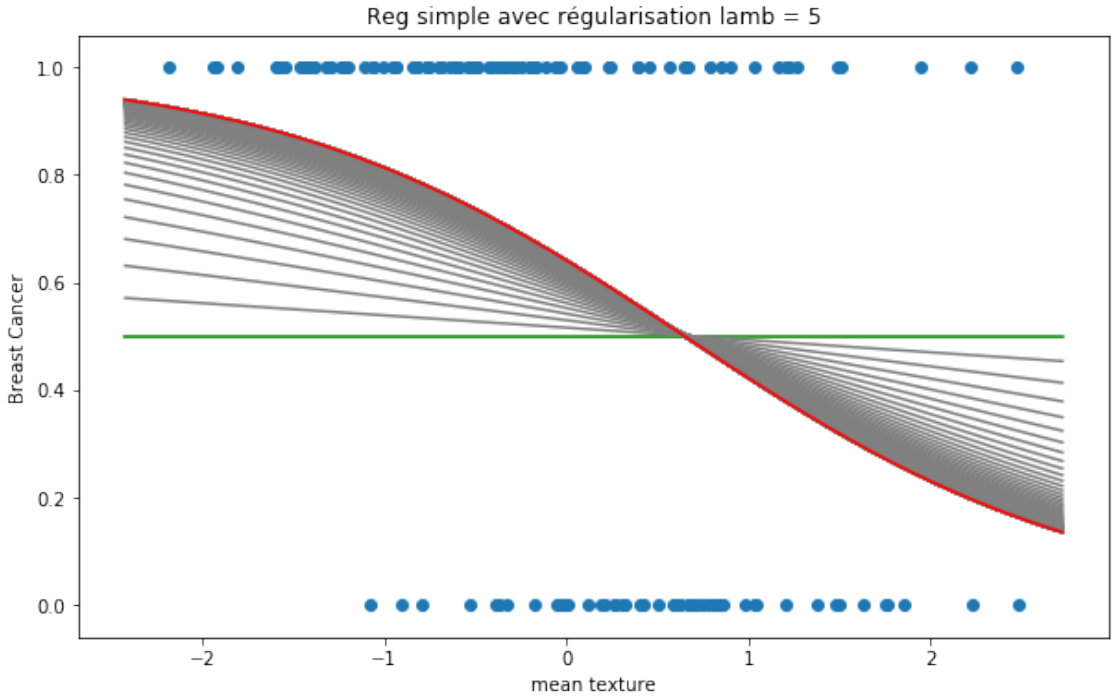


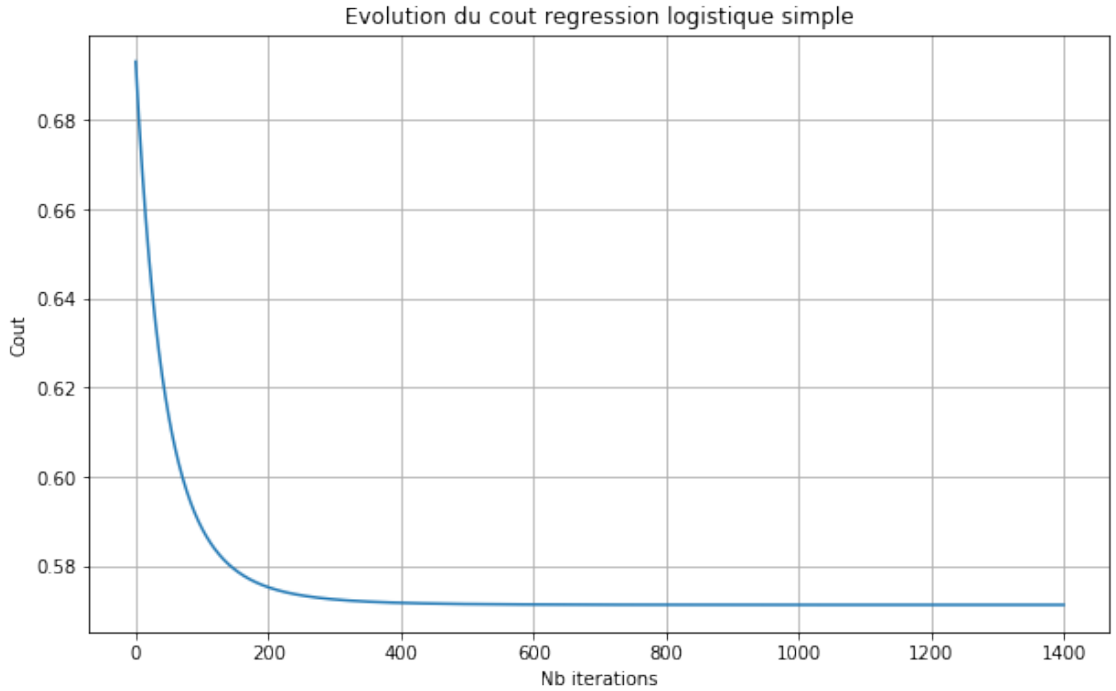

```
Betas : [0.6113945290041471, -0.9502822765761495]
```

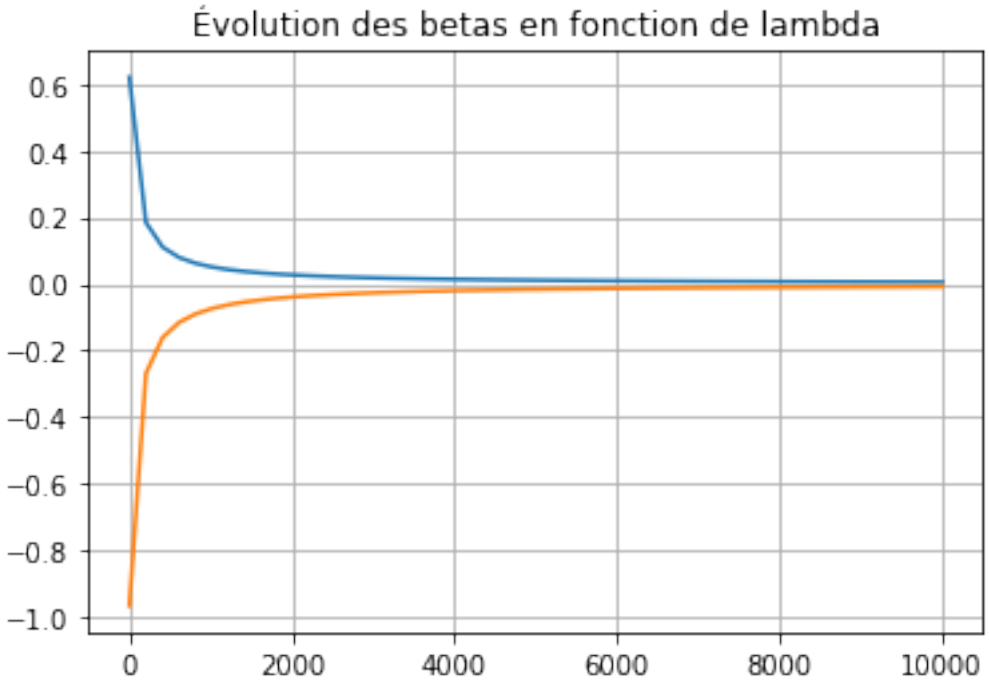


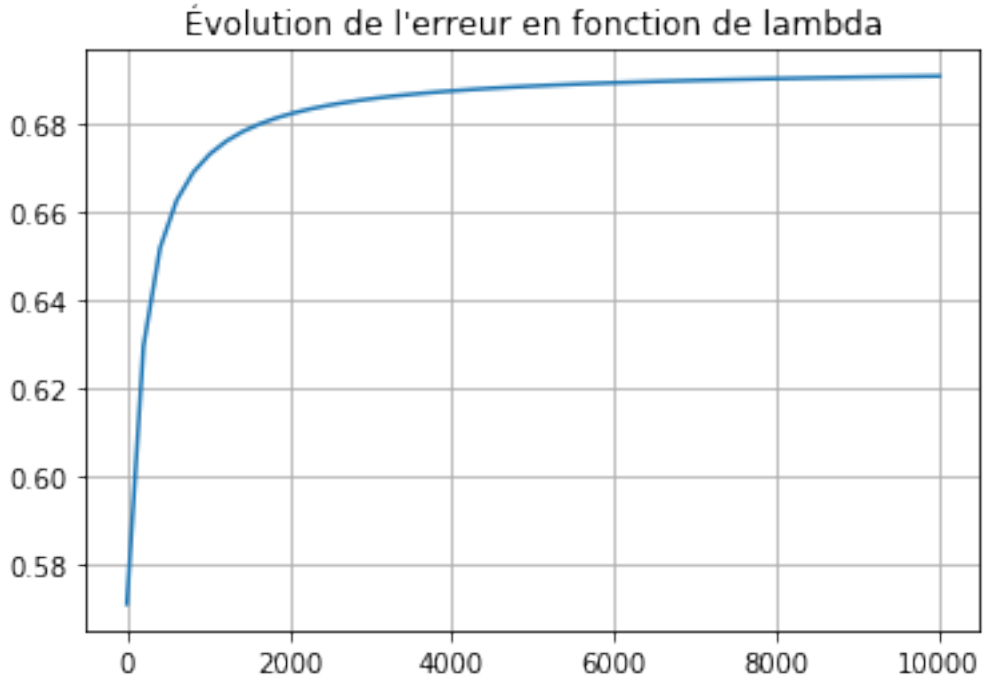



```
Betas : [0.5782868880351383, -0.8914578488948227]
```

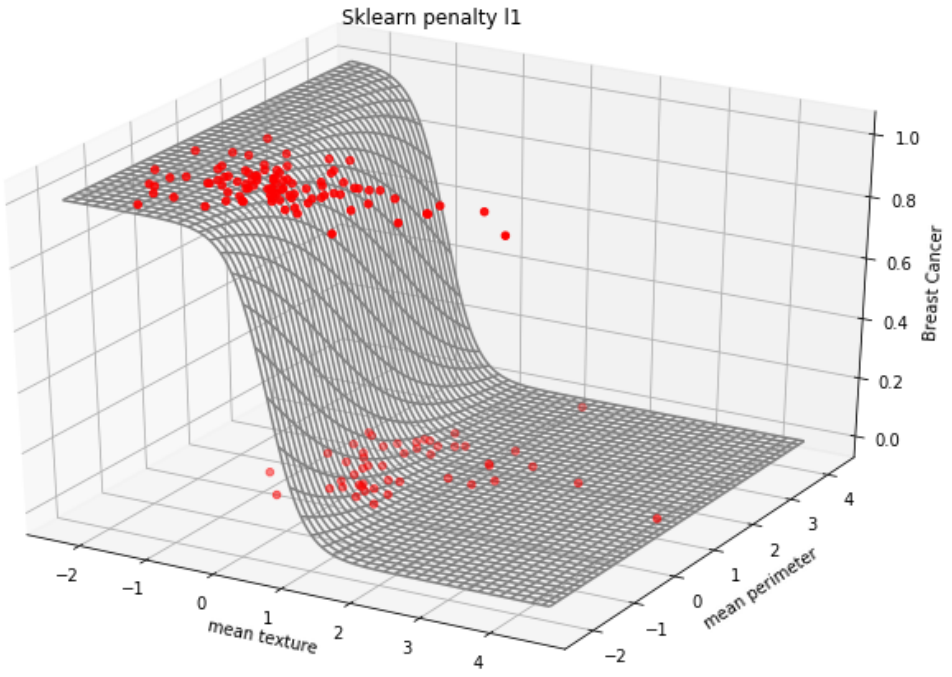

Reg simple avec régularisation lamb = 5
Breast Cancer
mean texture
1.0
0.8
0.6
0.4
0.2
0.0
−2
−1
0
1
2


Evolution du cout regression logistique simple
Cout
Nb iterations
0.68
0.66
0.64
0.62
0.60
0.58
0
200
400
600
800
1000
1200
1400

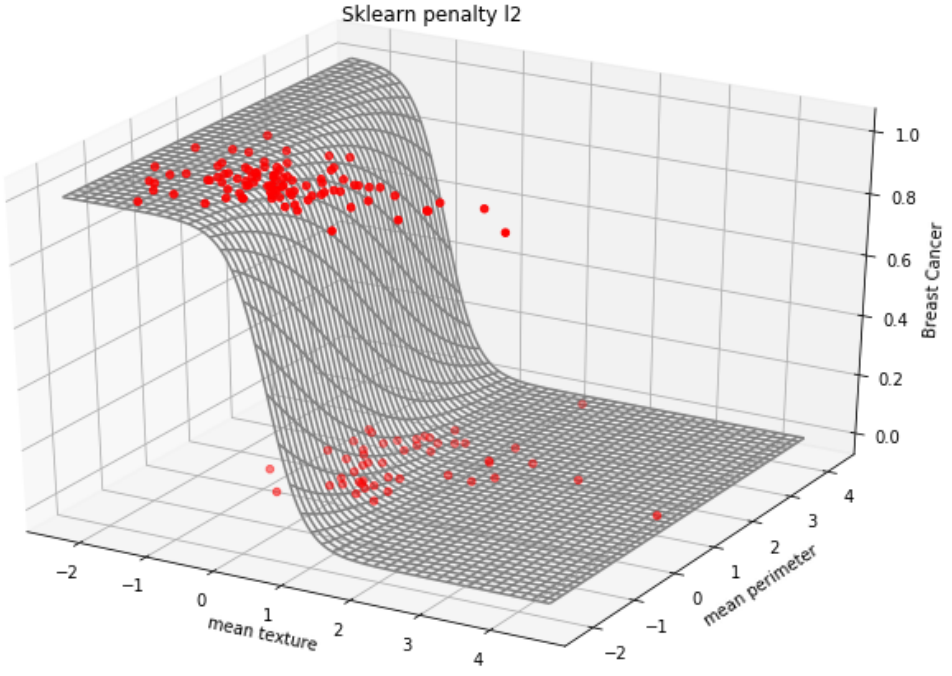




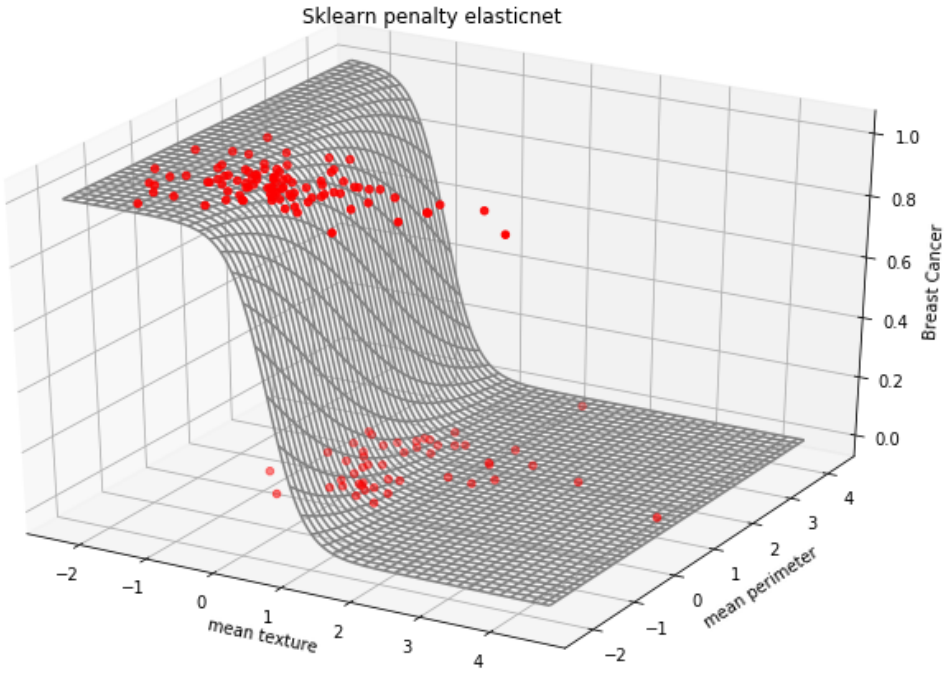

```
Betas : [ 0.27048095 -0.93529725 -4.23127392]
```




```
Betas : [ 0.27048703 -0.93523288 -4.23083155]
```



```
Betas : [ 0.27055507 -0.93528055 -4.23134262]
```



CM06_generalisation

May 11, 2020

1 Qualité de l'apprentissage

Ce notebook permet d'illustrer différents aspects de la qualité de l'apprentissage (biais-variance, sur et sous-apprentissage) en utilisant la régression polynomiale dans le contexte d'une variable explicative.
Nous nous intéressons pas à la théorie sous-jacente aux modèles de régression polynomiale.

Now its important to realise that in data science you will rarely use this kind of simple 2-dimensional modelling. The problem is stylised, and you rarely get simple Gaussian noise like this. In many ways this is a trivial, unrealistic example of learning and statistics. However, the simple nature of the material means we can carefully study the different aspects of learning, and discuss some of the behaviour that theory tells us about. So this is excellent material for a tutorial.

1.1 Régression polynomiale

On cherche à expliquer/prédire y à partir d'une seule variable explicative x. Par exple, un modèle de régression quadratique estimera y en utilisant la fonction f suivante :

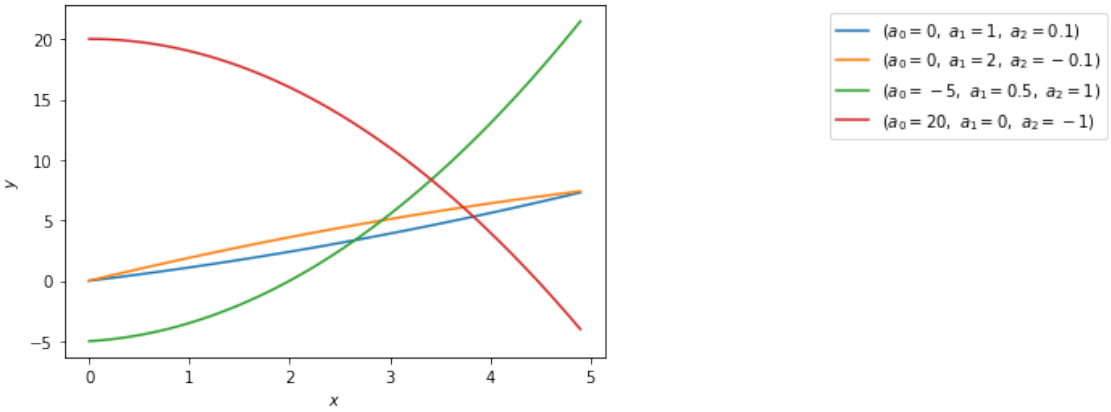
$$y \approx f(x) = a_0 \times 1 + a_1 \times x + a_2 \times x^2$$

où les trois paramètres à *apprendre* sont (a, b, c). La fonction f est donc une combinaison linéaire de *fonctions basiques* de x : $(1, x, x^2)$.

La Figure ci-dessous montre des courbes tracées pour différenres valeurs des paramètres (a, b, c).

[1]:
```
%matplotlib inline
import matplotlib.pyplot as pl
import numpy

# Générer des points entre (0,5)
x = numpy.arange(0., 5., 0.1)

def makelabel(a,b,c):
    return r'$(a_0=' + str(a0) + ',\ a_1=' + str(a1) + ',\ a_2=' + str(a2) +␣
 ↪')$'

# Courbes pour différentes valeurs des paramètres
[a0,a1,a2]  = [0,1,0.1]
pl.plot(x, a0 + a1*x + a2*x**2, label=makelabel(a0,a1,a2))
```

```
[a0,a1,a2]  = [0,2,-0.1]
pl.plot(x, a0 + a1*x + a2*x**2, label=makelabel(a0,a1,a2))
[a0,a1,a2]  = [-5,0.5,1]
pl.plot(x, a0 + a1*x + a2*x**2, label=makelabel(a0,a1,a2))
[a0,a1,a2]  = [20,0,-1]
pl.plot(x, a0 + a1*x + a2*x**2, label=makelabel(a0,a1,a2))
pl.legend(bbox_to_anchor=(1.4, 1.0))
pl.ylabel(r'$y$')
pl.xlabel(r'$x$')
pl.show()
```



Dans le cadre d'une régression polynomiale de degré n (aussi appelé *ordre*), on cherche $n+1$ coefficients tel que

$$y \approx f(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x_n = \sum_{i=0}^{n} a_i x^i$$

2 Construction d'un jeu de données synthétique

Nous allons construire ici un jeu de données synthétique pour lequel nous connaissons la fonction qui permet de générer de nouvelles données. L'avantage des données synthétiques est de pouvoir contrôler le différentes expérimentations, nottament le nombre de données (d'apprentissage et de test), et le degré des fonctions polynomiales testées.

2.1 Configuration

Le jeu de données comprends plusieurs charactéristiques, dont le nombre d'observations, la fonction qui permet de générer les données, le niveau de bruit, ou encore les valeurs minimum et maximum pour l'affichage.

Les valeurs de x sont tirées aléatoirement dans un intervalle donné pour un nombre d'observations données : + *rd.xmin* : minimum pour x de la bibliothèque "regressiondemo.py" fixé à 0 + *rd.xmax* : maximum for x, de la bibliothèque "regressiondemo.py", fixé à 10 + m : nombre de points/observations, défini localement dans le notebook

Les valeurs de y dépendent uniquement de la "vraie" fonction f et du niveau de bruit + *true-func()* : la "vraie" fonction f qui a permi de générer les données, défini localement dans le notebook + *sigma* : l'écart-type du bruit, défini localement dans le notebook

Pour l'affichage, on restreint les valeurs de y (par commodité) + *ydisplaymin* : minimum de y pour les figures, défini localement dans le notebook + *ydisplaymax* : maximum de y pour les figures, défini localement dans le notebook

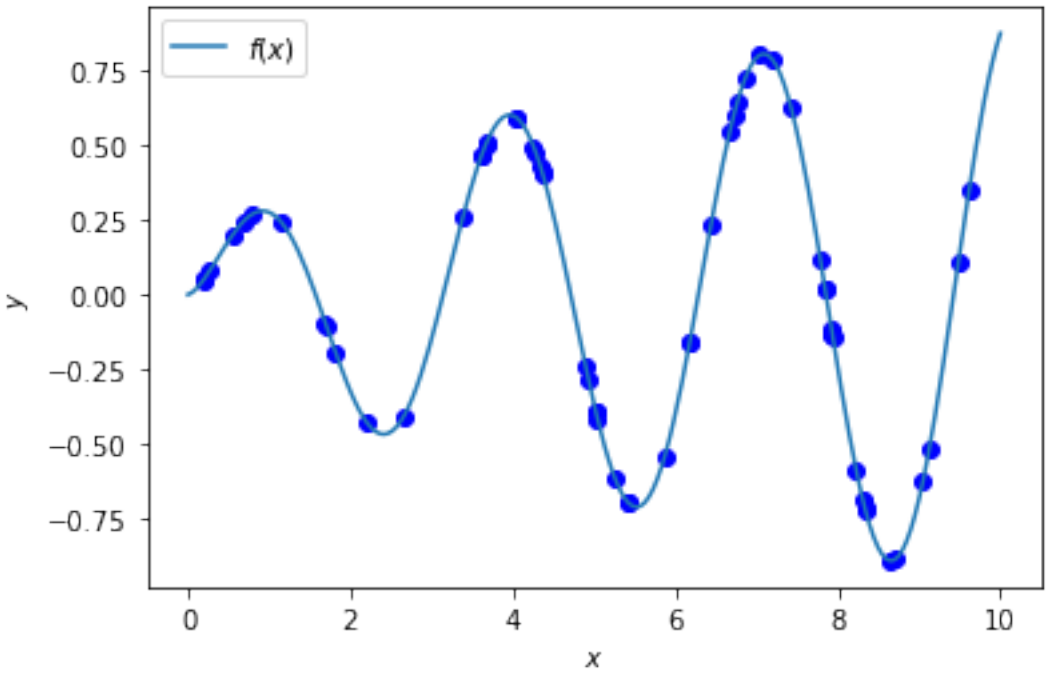
2.2 Etape 1.1 : génération de données non-bruitées

Dans un premier temps, nous générons des données sans bruit. Les poins sont donc parfaitement sur la fonction f. La fonction f, le nombre de points/observations (données d'apprentissage) et le niveau de bruit peuvent être modifées ci-dessous.

[2]:
```
# import
%matplotlib inline
import matplotlib.pyplot as pl
import sys
import os
#  charger la librairie fournie
sys.path.append(os.getcwd())
import regressiondemo as rd
```

[88]:
```
##############################
#      Configuration
#      =====================
# La "vraie" fonction (f(x))
def truefunc(x):
    return numpy.sin(x*2.0)*numpy.sqrt(x)/3.3

#  pour l'affichage
ydisplaymin = -1.8
ydisplaymax = 1.8

#  ne testez pas avec plus de 100 points/observations O(m^3)
m = 60

#  niveau de bruit
rd.setSigma(0.2)

# génération d'un vecteur x (aléatoire)
x = rd.makeX(m)
# les vraies valeurs de y
yt = truefunc(x)

# géénré xts et yts pour l'affichage (forte densité de point)
xts = rd.makeX(200,uniform=True)
yts = truefunc(xts)
```

[89]:
```
# Figure
pl.plot(x, yt, 'bo')
pl.plot(xts, yts,label = r'$f(x)$')
pl.ylabel(r'$y$')
pl.xlabel(r'$x$')
pl.legend()
pl.show()
```



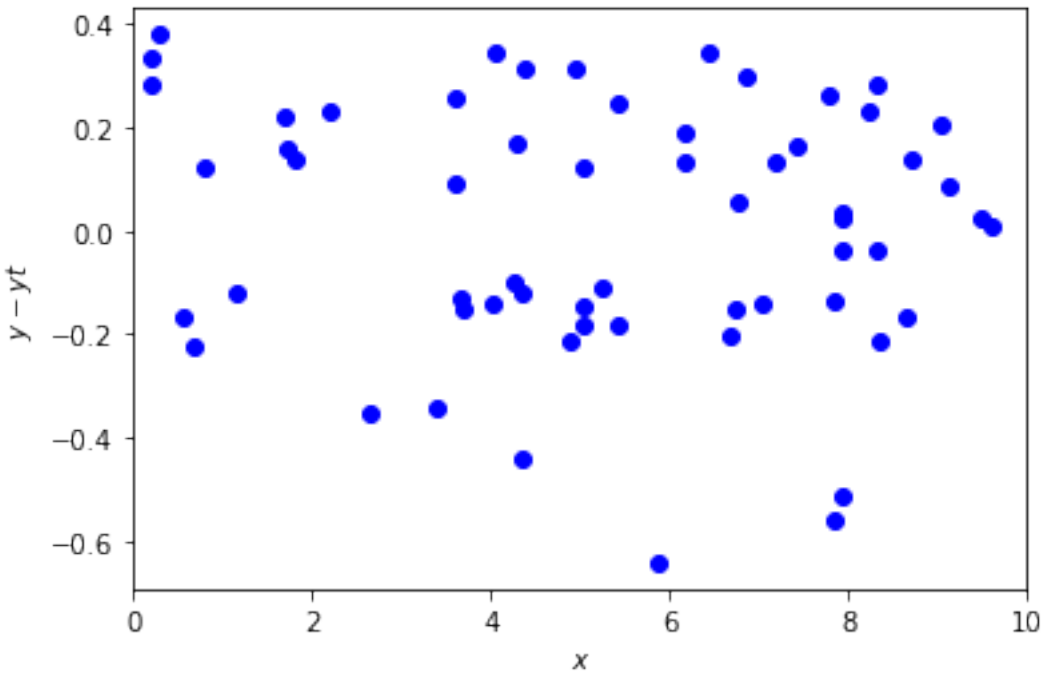
2.3 Etape 1.2 : ajouter du bruit

Afin d'obtenir des données plus réalistes, une approche courante consiste à ajouterdu bruit à y. Ici, on jaoute un bruit blanc Gaussien d'écart-type *sigma*.

Que représente d'après-vous la première Figure ?

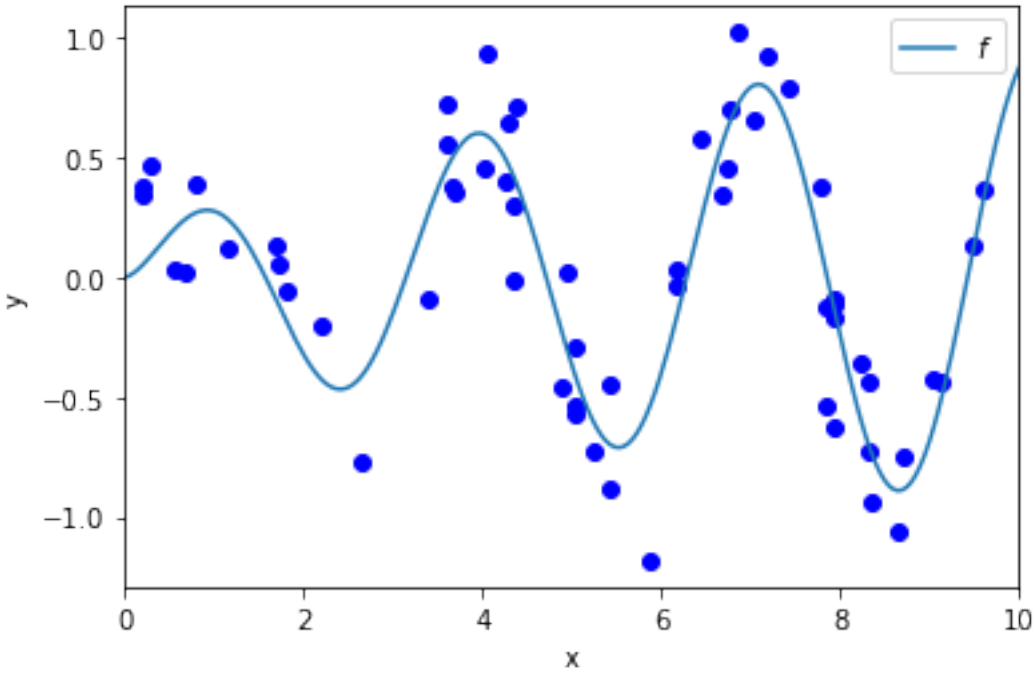
[90]:
```
#  Ajouter le bruit Gaussien
y = rd.addNoise(yt)
```

[91]:
```
pl.plot(x, y-yt, 'bo')
pl.ylabel(r'$y-yt$')
pl.xlabel(r'$x$')
pl.xlim(rd.xmin,rd.xmax)
pl.show()
```



[92]:
```
# Données
pl.plot(x, y, 'bo')
pl.ylabel('y')
pl.xlabel('x')
pl.xlim(rd.xmin,rd.xmax)
pl.plot(xts, yts,label = r'$f$')
pl.legend()
```

[92]: <matplotlib.legend.Legend at 0x282f523a3c8>



3 Régression polynomiale

Nous allons utiliser la fonction de régression polynomiale de Numpy *numpy.polyfit()*.
Cette dernière cherche à déterminer les coefficients a_i ($1 \leq i \leq n$), avec n le degré de la fonction polynomiale, qui permettent de minimiser le coût quadratique (même coût que la régression linéaire) :

$$\sum_{i=1}^{m}(y_i - f(x_i))^2$$

On ne s'intéresse pas ici à l'algorithme de résolution (noté tout de même que l'on pourrait utilisé l'algorithme de descente de gradient.).

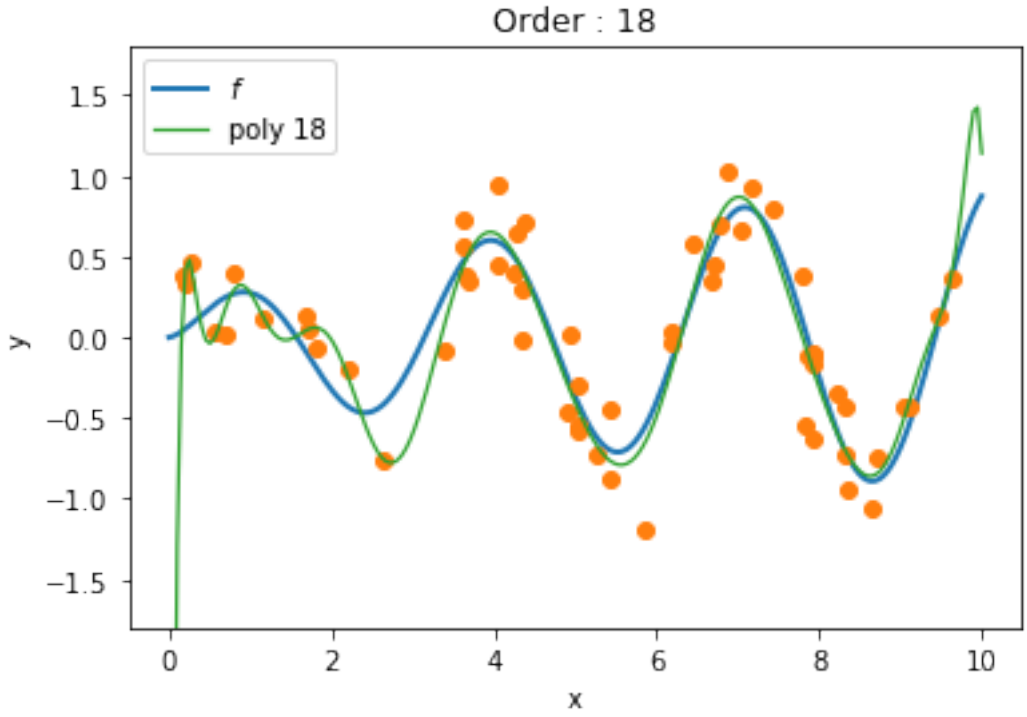
3.1 Etape 1.3 : régression polynomiale

Testons dans un premiers temps un polynome d'ordre 4. Notez que la courbe bleue épaisse est pour f tandis que la courbe verte plus fine corresponds à la fonction apprise par le modèle $\hat{f}$.

[103]:
```
##############################
#      Configuration
#      =====================
#     changer la valeur de *order* pour tester des régressions polynomiales de␣
 ↪différents degrés
order = 18
##############################
```

[104]:
```
# plot data and the truth
pl.plot(xts, yts,label = r'$f$', linewidth=2)
title = "Order : "+str(order)
pl.title(title)
pl.plot(x, y, 'o')
pl.ylabel('y')
pl.xlabel('x')

#  build the fitted poly curve (xts,ys) from order-th regression
ys = rd.linReg(x,y,xts,order)

#  plot fitted curve
pl.plot(xts, ys,label = 'poly ' + str(order), linewidth=1.25 )
pl.ylim(ydisplaymin,ydisplaymax)
pl.legend()
```

[104]:
```
<matplotlib.legend.Legend at 0x282f4fc5f98>
```



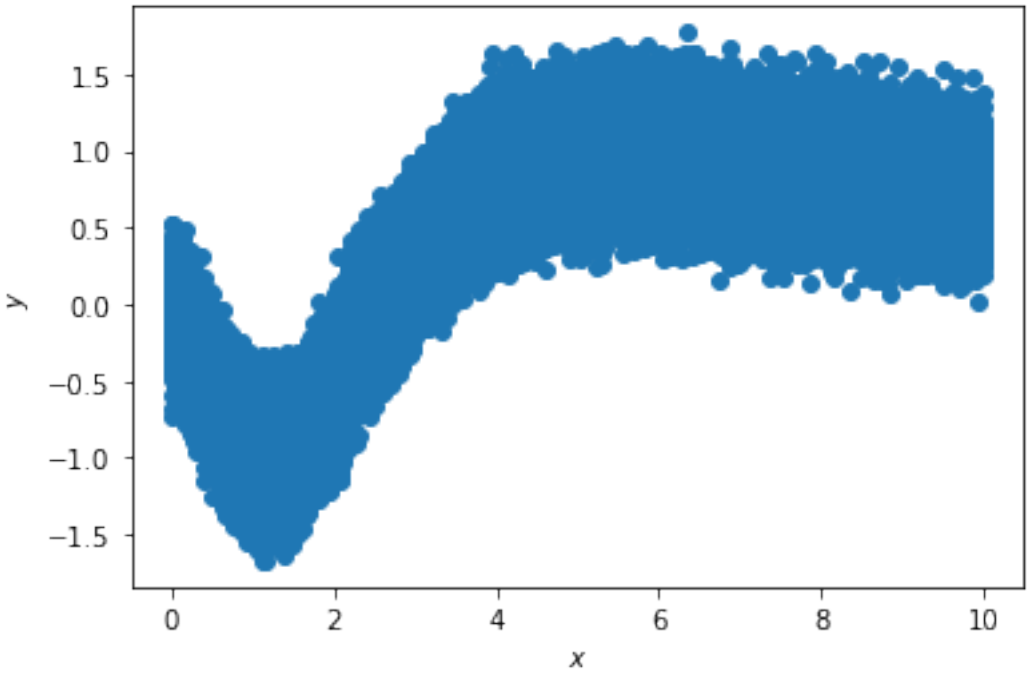
Les régressions polynomiales de degré bas ne marchent pas très bien pour expliquer notre fonction f complexe.

3.2 Etape 1.4 : Sur une fonction mystère

Nous générons ici de nouvelles données. La fonction *rd.demoReg1()* défini sa propre fonction f pour générer les données.

[125]:
```
##############################
#      Configuration
#      =====================
#     # observations/points
m = 100000
##############################

x,y = rd.demoReg1(m)
# plot
pl.plot(x, y, 'o')
pl.ylabel(r'$y$')
pl.xlabel(r'$x$')
```

[125]:
```
Text(0.5, 0, '$x$')
```

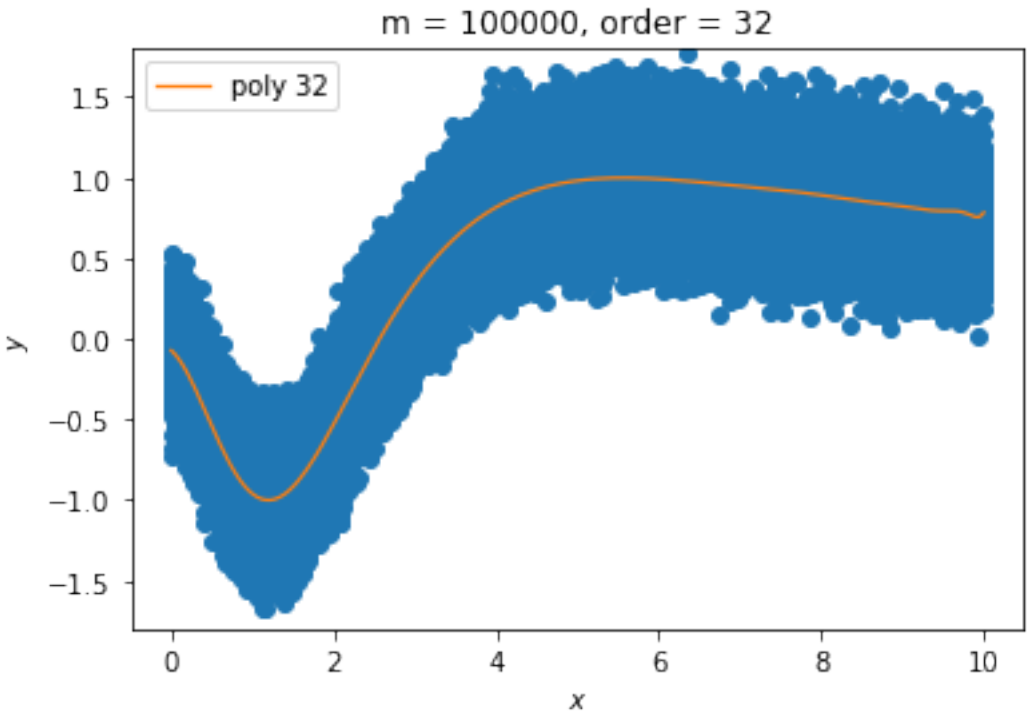


Nous allons maintenant testé différents ajustements

[131]:
```
##############################
#      Configuration
#      =====================
#     changer la valeur de *order* pour tester des régressions polynomiales de␣
 ↪différents degrés
order = 32
##############################
#  ajustement
ys = rd.linReg(x,y,xts,order)
```

```
# plot
pl.plot(x, y, 'o')
pl.title("m = "+str(m)+", order = "+str(order))
pl.ylabel(r'$y$')
pl.xlabel(r'$x$')
pl.plot(xts, ys,label = 'poly ' + str(order), linewidth=1.25 )
pl.ylim(ydisplaymin,ydisplaymax)
pl.legend()
```

[131]: `<matplotlib.legend.Legend at 0x282800102b0>`



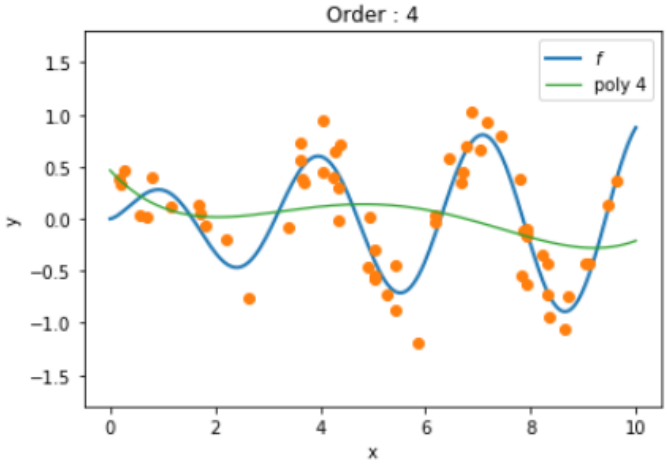
Testez pour différentes quantités de données d'apprentissage (m) différents types d'ajustements (*order*) et différents types de bruit.

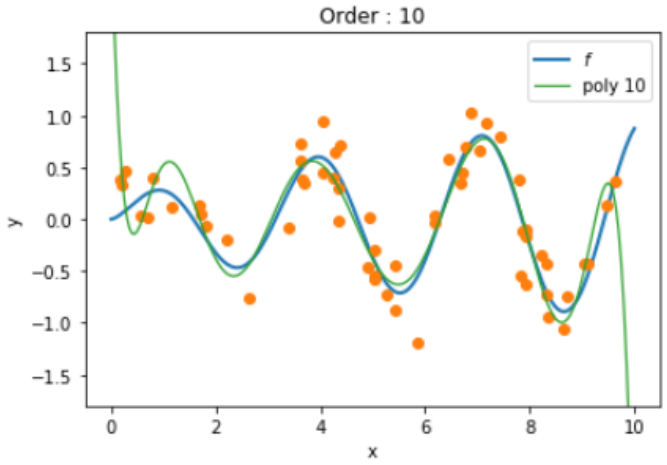
Ecrivez un document résumant sur vos différentes observations.

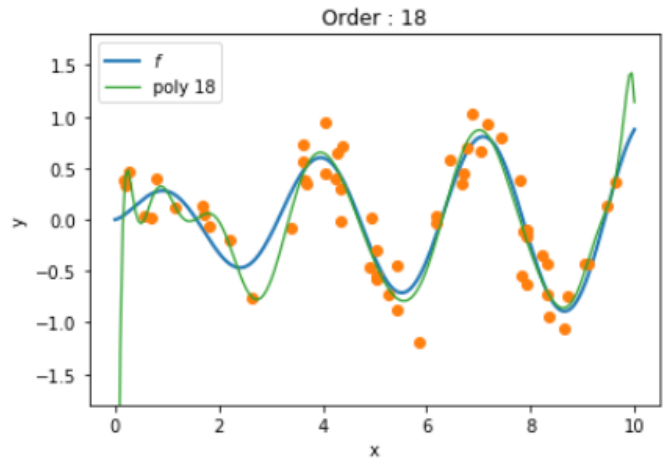
Régression polynomiale

Avec une fonction connue

Pour le nombre d'observations m = 60 :



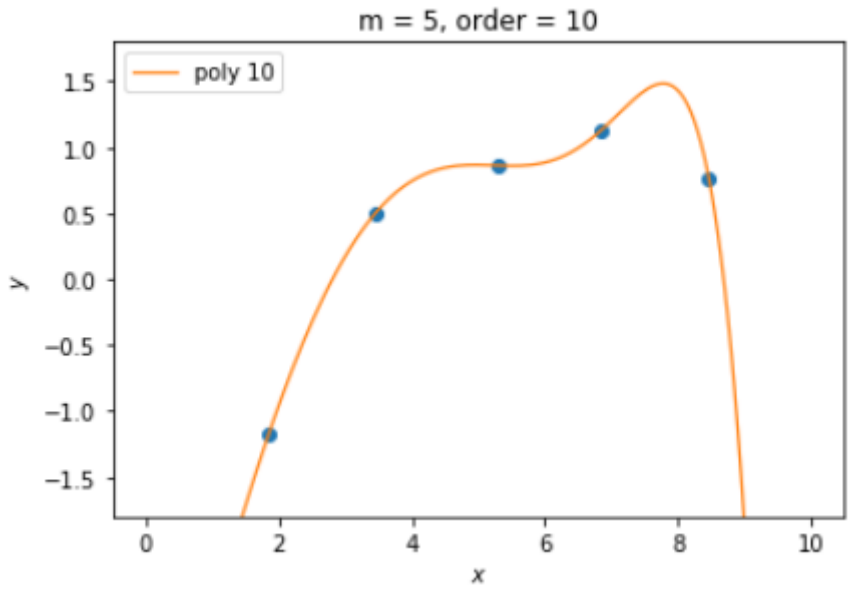




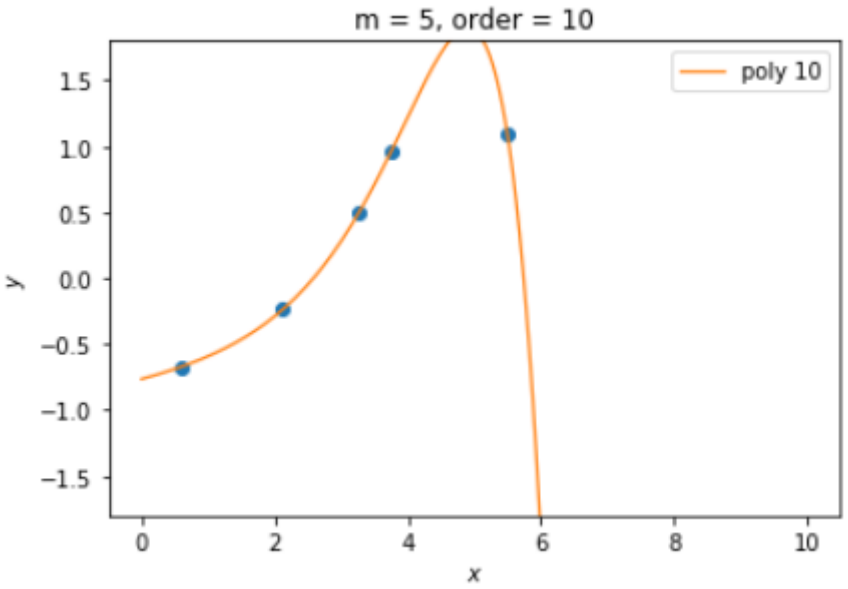
On remarque qu'avec un ordre faible, la courbe calculée ne ressemble vraiment pas à la fonction connue. A l'inverse si l'ordre est trop élevé, la courbe va être trop dépendante des données réelles et donc possiblement du bruit. Elle est à haute variance.

Avec une fonction mystère

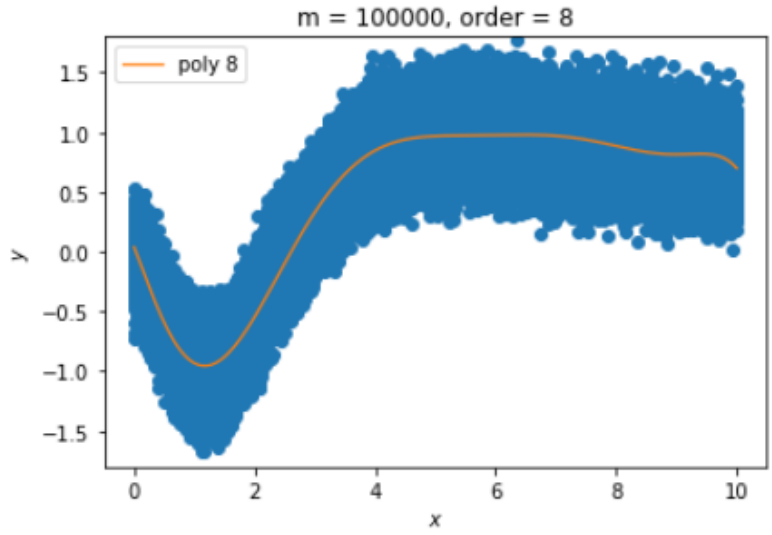
Test n°1 :

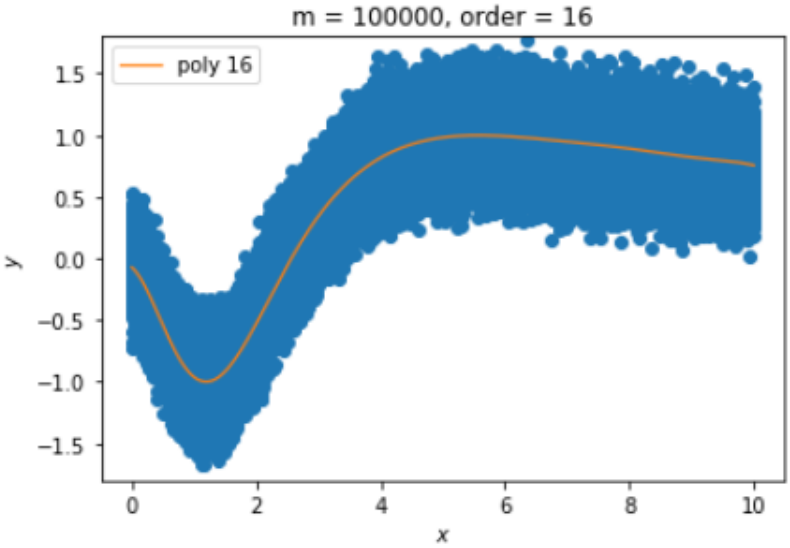


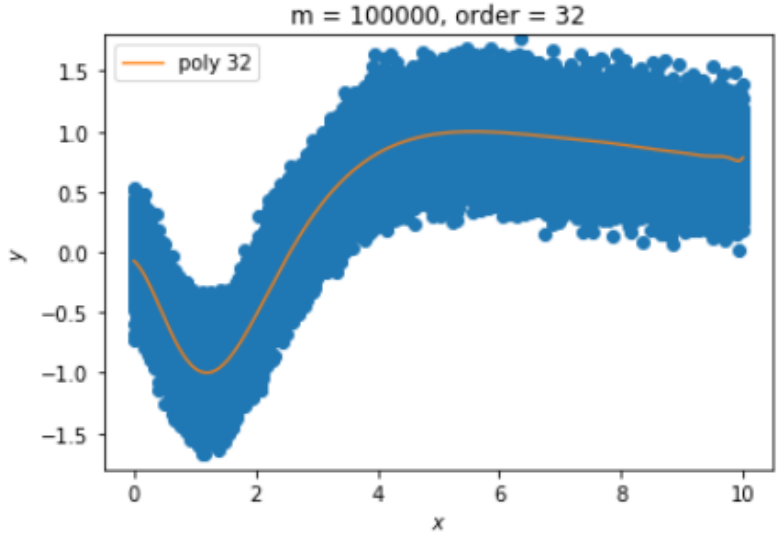
Test n°2 :



On remarque que si le nombre d'observation est faible, la courbe résultante est fortement impactée par le bruit.


m = 100000, order = 8
poly 8
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
0
2
4
6
8
10
x


m = 100000, order = 16
poly 16
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
0
2
4
6
8
10
x


m = 100000, order = 32
poly 32
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
0
2
4
6
8
10
x

En augmentant significativement le nombre d'observations, on observe que l'ordre impacte peu la tendance de la courbe.

Conclusion

Il faut un nombre d'observations le plus élevé possible. L'ordre ne doit pas être trop faible. Néanmoins s'il est trop élevé la variance de la courbe sera forte. Si l'ordre est trop faible, le biais de cette courbe sera fort.

Il faut donc trouver un compromis entre le biais et la variance afin de minimiser ces 2 variables. C'est le contrepartie biais-variance.

TP7_classif

May 11, 2020

1 Classification : Bayésien naïf, LDA, QDA, et k-Plus Proches Voisins

L'objectif de cette séance de TD/TP est 1. d'implémenter les différents algorithmes de classificaion vus en cours * bayésien naïf * LDA et QDA * k-PPV 2. d'évaluer les performances de classification sur *les échantillons d'apprentissage* * calcul d'une matrice de confusion * calcul de l'Overall Accuracy et du coefficient Kappa 3. de visualiser les frontières de décision obtenues par les algorithmes de classification

L'ensemble des algorithmes à implémenter sont déterministiques (c'est-à-dire qu'ils ne dépendent pas d'un caractère aléatoire). Vos implémentations devraient donc théoriquement vous donner des résultats strictement identiques à ceux de Scikit-Learn. Cependant, la résolution de LDA par Scikit-Learn n'est pas exacte et l'algorithme des k-PPV peut utiliser de l'aléatoire pour départager les cas d'égalité lors de l'affectation de la classe majoritaire (diapo 32 du cours).

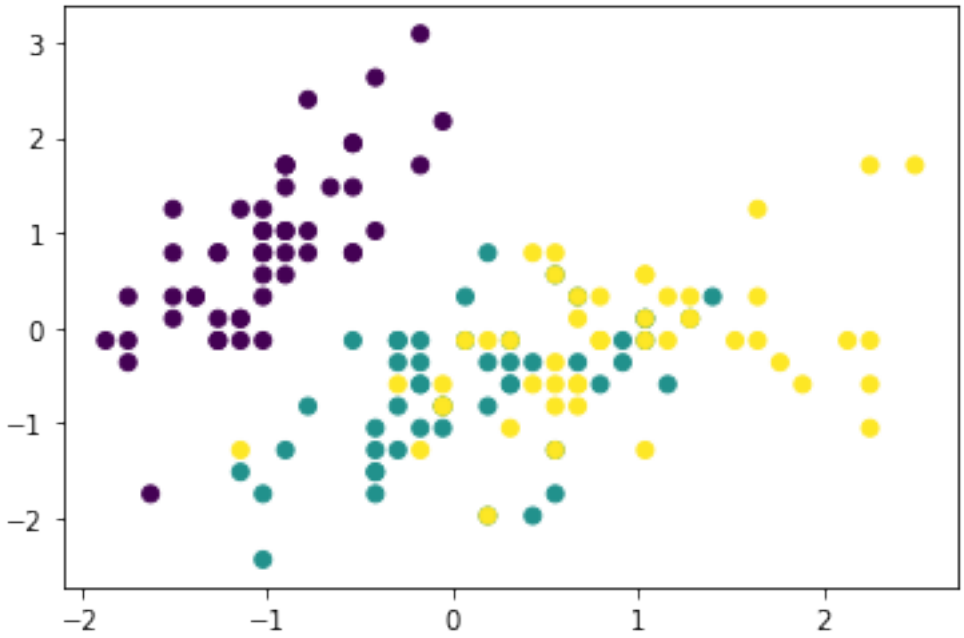
Indication : le découpage en fonction (avec les résultats attendus) est là pour vous aider à modulariser votre code. Cependant, vous pouvez partir sur un tout autre choix d'implémentation (notamment d'autres paradigmes de programmation comme l'orienté objet ou le fonctionnel).

1.1 Jeu de données

[30]:
```
#-- import
from sklearn import datasets
import math
import numpy as np
from scipy import stats
import random
import matplotlib.pyplot as plt

from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.neighbors import KNeighborsClassifier
from sklearn import linear_model

from sklearn.metrics import accuracy_score
from sklearn.metrics import cohen_kappa_score
from sklearn.metrics.pairwise import euclidean_distances #to compute pairwise␣
↪distances
from numpy.linalg import inv #to compute X^-1
```

[31]:
```
#-- Charger le jeu de données iris
iris = datasets.load_iris()
X = iris.data[:,0:2] # on garde uniquement deux variables explicatives
X = (X-X.mean(axis=0))/X.std(axis=0,ddof=0)
y = iris.target
```

[32]:
```
#-- Scatterplot
plt.figure()
plt.scatter(X[:,0],X[:,1], c=y) #plot the points with their label
plt.show()
```



1.2 Classifieur bayésien naïf

1.2.1 1. Algotihme de Scikit-learn

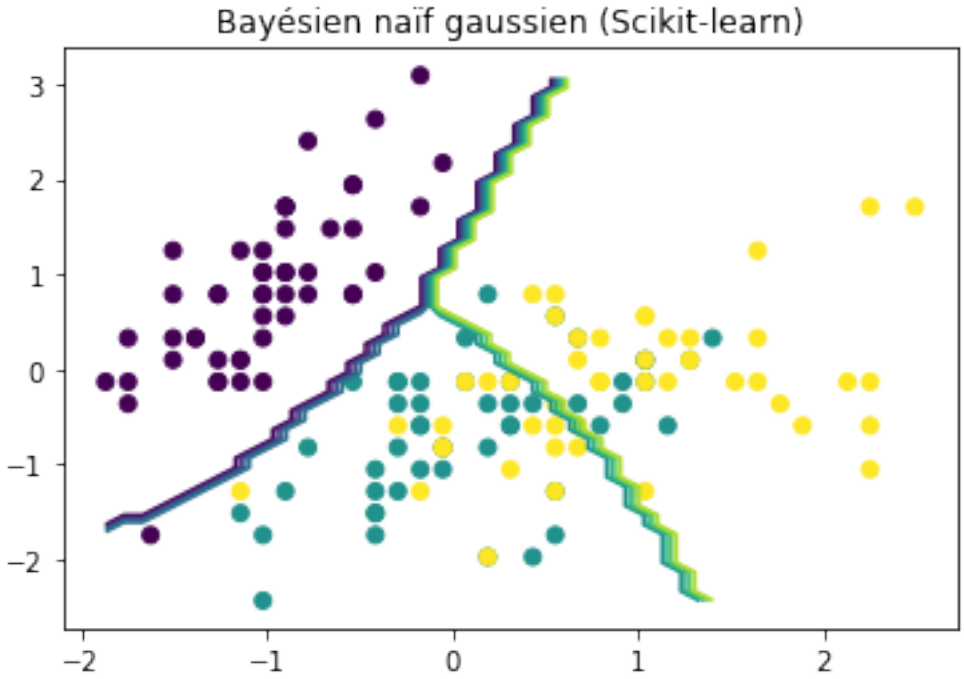
[33]:
```
#-- Trouver le nombre de classes
C = len(np.unique(y))
print("C =", C)
```

```
C = 3
```

[34]:
```
#-- Bayésien naif gaussien
gnb = GaussianNB()
y_pred = gnb.fit(X, y).predict(X)
print ("Taux d'erreur du classifieur bayésien naïf", np.mean(y_pred != y))
```

```
Taux d'erreur du classifieur bayésien naïf 0.22
```

[35]:
```
#-- Figure (frontières de décision)
x_min, x_max, y_min, y_max = np.min(X[:, 0]),np.max(X[:, 0]) ,  np.min(X[:,␣
↪1]),  np.max(X[:, 1])
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),np.arange(y_min, y_max, 0.1))
Z = gnb.fit(X, y).predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.figure()
plt.scatter(X[:,0],X[:,1], c=y)
plt.contour(xx, yy, Z)
plt.title("Bayésien naïf gaussien (Scikit-learn)")
plt.show()
```



1.2.2 2. Votre version de l'algorithme Bayésien naif

Astuces : * `np.bincount(y)` permet de compter le nombre d'occurences dans une classe * `np.mean` et `np.var` permettent de calculer la moyenne et la variance pour une variable explicative * loi normale pour z la variable aléatoire : $p(z) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(z-\mu)^2}$ * `grid = np.c_[xx.ravel(),yy.ravel()]` défini une matrice avec toutes les positions sur une grille (cf. le code de la Figure ci-dessus)

[36]:
```
#-- probabilité a priori (Pr(Y=y))
def proba_classe(y):
    return np.bincount(y) / y.shape[0]
```

[37]:
```
print(proba_classe(y)) #- fréquence d'apparition de chacune des classes dans␣
 ↪les données d'apprentissage
```

```
[0.33333333 0.33333333 0.33333333]
```

Résultat attendu [0.33333333 0.33333333 0.33333333]

[52]:
```
def estimation_param_nb(X,y):
    nbClasses = proba_classe(y).shape[0]
    mu = []
    sigma2 = []
    for i in range(nbClasses):
        classe =[]
        for j in range(y.shape[0]):
            if y[j] == i:
                classe.append(X[j])
        mu.append(np.mean(classe,axis=0))
        sigma2.append(np.var(classe,axis=0))
    return (np.asarray(mu),np.asarray(sigma2))
```

[49]:
```
print(estimation_param_nb(X,y))
```

```
(array([[-1.01457897,  0.85326268],
       [ 0.11228223, -0.66143204],
       [ 0.90229674, -0.19183064]]), array([[0.17876968, 0.74619175],
       [0.38334383, 0.51135882],
       [0.5817693 , 0.54010089]]))
```

Résultat attendu (array([[-1.01457897, 0.85326268], [0.11228223, -0.66143204], [0.90229674, -0.19183064]]), array([[0.17876968, 0.74619175], [0.38334383, 0.51135882], [0.5817693 , 0.54010089]]))

[77]:
```
#-- loi normale (1D)
def loi_normale(x,mu,sigma2):
    return np.exp(-(x-mu)**2 / (2*sigma2)) / np.sqrt(2*np.pi*sigma2)
```

[78]:
```
loi_normale(X[:,0],mu=0,sigma2=1)[:10]
```

[78]:
```
array([0.26592211, 0.20759165, 0.15281312, 0.12825426, 0.23668473,
       0.34534257, 0.12825426, 0.23668473, 0.08645012, 0.20759165])
```

Résultat attendu array([0.26592211, 0.20759165, 0.15281312, 0.12825426, 0.23668473, 0.34534257, 0.12825426, 0.23668473, 0.08645012, 0.20759165])

[50]:
```
#-- vraisemblance (Pr(X=x|Y=y)) en fonction des paramètres mu et sigma
def vraisemblance_nb(X,mu,sigma2):
    N, d = X.shape
    C = mu.shape[0]
    pvrais = np.zeros((N,C))
```

```
    for c in range(C):
        norm = loi_normale(X,mu[c],sigma2[c])
        for dim in range(N):
            pvrais[dim][c] = np.prod(norm[dim])
    return pvrais
```

[308]:
```
mu, sigma2 = estimation_param_nb(X,y)
print(vraisemblance_nb(X,mu,sigma2)[:10,:])
```

```
[[4.12568710e-01 5.96043661e-03 4.47104459e-03]
 [2.17134544e-01 3.49960316e-02 7.76844498e-03]
 [2.46660743e-01 7.39752504e-03 2.46053810e-03]
 [1.51142680e-01 6.70198548e-03 1.79316099e-03]
 [3.92250378e-01 1.89179325e-03 1.72350876e-03]
 [1.04438709e-01 2.77596442e-04 7.12783062e-04]
 [2.20840906e-01 1.50708277e-03 7.95813815e-04]
 [4.34485017e-01 8.58896773e-03 4.83801148e-03]
 [3.58456742e-02 3.59355840e-03 6.57792476e-04]
 [2.83993851e-01 2.61821854e-02 7.21025739e-03]]
```

Résultat attendu [[4.12568710e-01 5.96043661e-03 4.47104459e-03] [2.17134544e-01 3.49960316e-02 7.76844498e-03] [2.46660743e-01 7.39752504e-03 2.46053810e-03] [1.51142680e-01 6.70198548e-03 1.79316099e-03] [3.92250378e-01 1.89179325e-03 1.72350876e-03] [1.04438709e-01 2.77596442e-04 7.12783062e-04] [2.20840906e-01 1.50708277e-03 7.95813815e-04] [4.34485017e-01 8.58896773e-03 4.83801148e-03] [3.58456742e-02 3.59355840e-03 6.57792476e-04] [2.83993851e-01 2.61821854e-02 7.21025739e-03]]

[44]:
```
#-- classifieur bayésien naif (NB : Naif Bayesian classifier)
def predict_nb(X,Xtrain,ytrain):
    """
        Prédiction des étiquettes pour les échantillons X lorsque le modèle est␣
 ↪appris sur les données (Xtrain,ytrain)
        IN:
            - X (N,d) : observations pour laquelle on souhaite prédire la␣
 ↪classe yhat
            - Xtrain (m,d) : données d'apprentissage
            - ytrain (m,): étiqueetes associées aux données d'apprentissage
        OUT:
            - yhat (N,) : classes prédites pour les observations X
    """
    mu, sigma2 = estimation_param_nb(Xtrain,ytrain)
    yhat = np.argmax(proba_classe(ytrain) * vraisemblance_nb(X, mu, sigma2),␣
 ↪axis=1)
    return yhat
```

[80]:
```
print(predict_nb(X,X,y)) # prédictions sur les données d'apprentissage (donc␣
 ↪X=Xtrain dans ce TP)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 1 0 0 0 0 0 0 0 0 2 2 2 1 2 1 2 1 2 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1
```

```
 2 2 2 2 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 1 2 2 1 2 2 2 2
 1 2 1 1 2 2 2 2 1 2 1 2 1 2 2 1 1 1 2 2 2 1 1 1 2 2 2 1 2 2 2 1 2 2 2 1 2
 2 1]
```

RESULTAT ATTENDU [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 2 2 2 1 2 1 2 1 2 1 1 1 1 1 2 1 1 1 1 1 1 2 2 2 2 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 2 1 2 1 2 2 1 2 2 2 2 1 2 1 1 2 2 2 2 1 2 1 2 1 2 2 1 1 1 2 2 2 1 1 1 2 2 2 1 2 2 2 1 2 2 2 1 2 2 1]

1.2.3 3. Evaluation des performances

[166]:
```python
#-- Calcul de la matrice de confusion
def getConfusionMatrix(y,yhat,C=None):
    if C is None:
        C = proba_classe(y).shape[0]
    confMatrix = np.zeros((C,C))
    for i in range(y.shape[0]):
        confMatrix[y[i]][yhat[i]] += 1
    return confMatrix
```

[110]:
```python
#-- Calcul du taux de bonne classification (OA)
def getOA(y,yhat, C=None):
    if C is None:
        C = getConfusionMatrix(y,yhat)
    return np.sum(C.diagonal()) / y.shape[0]
```

[171]:
```python
#-- Calcul du coefficient Kappa
def getKappa(y,yhat):
    C = proba_classe(y).shape[0]
    confMatrix = getConfusionMatrix(y,yhat,C=C)
    OA = getOA(y,yhat,C=confMatrix)
    sumT = 0
    for i in range(C):
        sum1 = 0
        for j in range(C):
            sum1 += confMatrix[i][j]
        sum2 = 0
        for j in range(C):
            sum2 += confMatrix[j][i]
        sumT += sum1 * sum2
    pn = sumT / (y.shape[0]**2)
    return (OA-pn) / (1-pn)
```

[172]:
```python
#-- test
yhat = predict_nb(X,X,y)
print ("Taux d'erreur du classifieur bayésien naïf", 1-getOA(y,yhat))
print ("\tet coefficient Kappa", getKappa(y,yhat))
```

```
Taux d'erreur du classifieur bayésien naïf 0.21999999999999997
        et coefficient Kappa 0.67
```

[113]:
```
print ("Taux d'erreur du classifieur bayésien naïf (Scikit-learn)",␣
↪1-accuracy_score(y,yhat))
print ("\tet coefficient Kappa (Scikit-learn)", cohen_kappa_score(y,yhat))
```

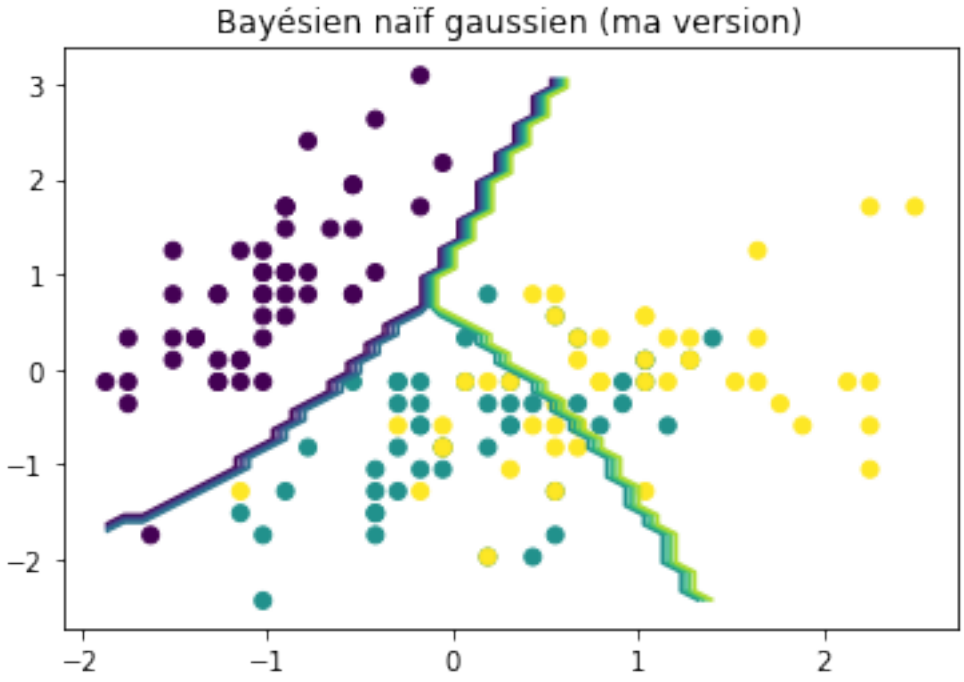
```
Taux d'erreur du classifieur bayésien naïf (Scikit-learn) 0.21999999999999997
        et coefficient Kappa (Scikit-learn) 0.6699999999999999
```

Résultat attendu Taux d'erreur du classifieur bayésien naïf 0.21999999999999997 et coefficient Kappa 0.67

Taux d'erreur du classifieur bayésien naïf (Scikit-learn) 0.21999999999999997 et coefficient Kappa (Scikit-learn) 0.6699999999999999

1.2.4 4. Visualisation des frontières de décision

[174]:
```
#-- Figure (frontières de décision)
## A COMPARER AVEC LE PLOT INITIAL
x_min, x_max, y_min, y_max = np.min(X[:, 0]),np.max(X[:, 0]) ,  np.min(X[:,␣
↪1]),  np.max(X[:, 1])
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),np.arange(y_min, y_max, 0.1))
Z = predict_nb(np.c_[xx.ravel(), yy.ravel()],X,y)
Z = Z.reshape(xx.shape)

plt.figure()
plt.scatter(X[:,0],X[:,1], c=y)
plt.contour(xx, yy, Z)
plt.title("Bayésien naïf gaussien (ma version)")
plt.show()
```



Les 2 plots sont similaires.

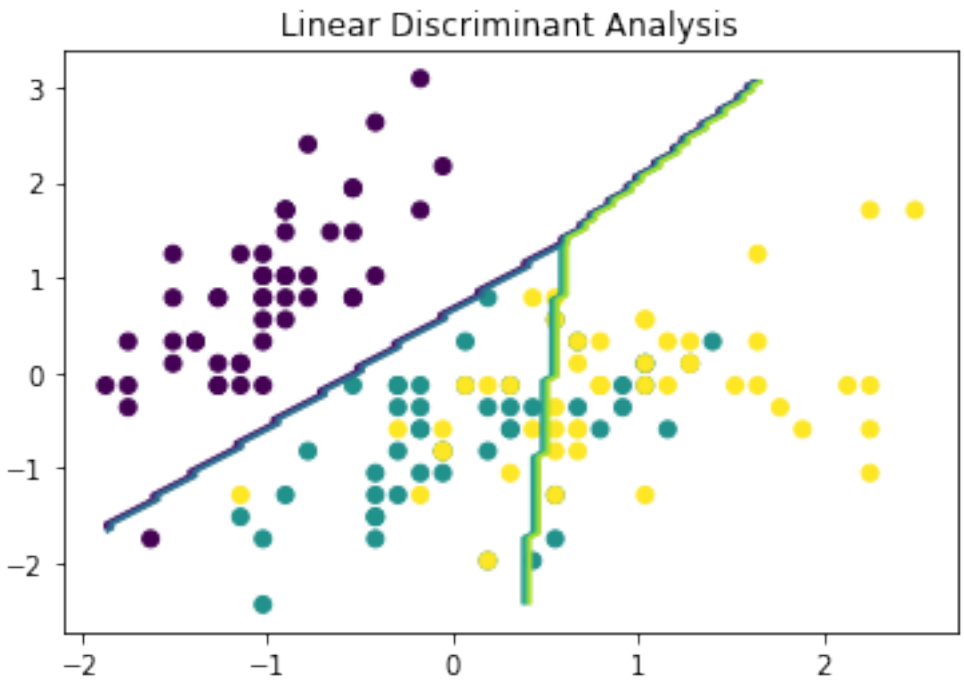
1.3 LDA et QDA

1.3.1 1. Scikit-learn

[175]:
```
#-- LDA
lda = LinearDiscriminantAnalysis()
lda.fit(X, y)
y_pred = lda.predict(X)
print ("Taux d'erreur LDA", np.mean(y_pred != y))
```

```
Taux d'erreur LDA 0.2
```

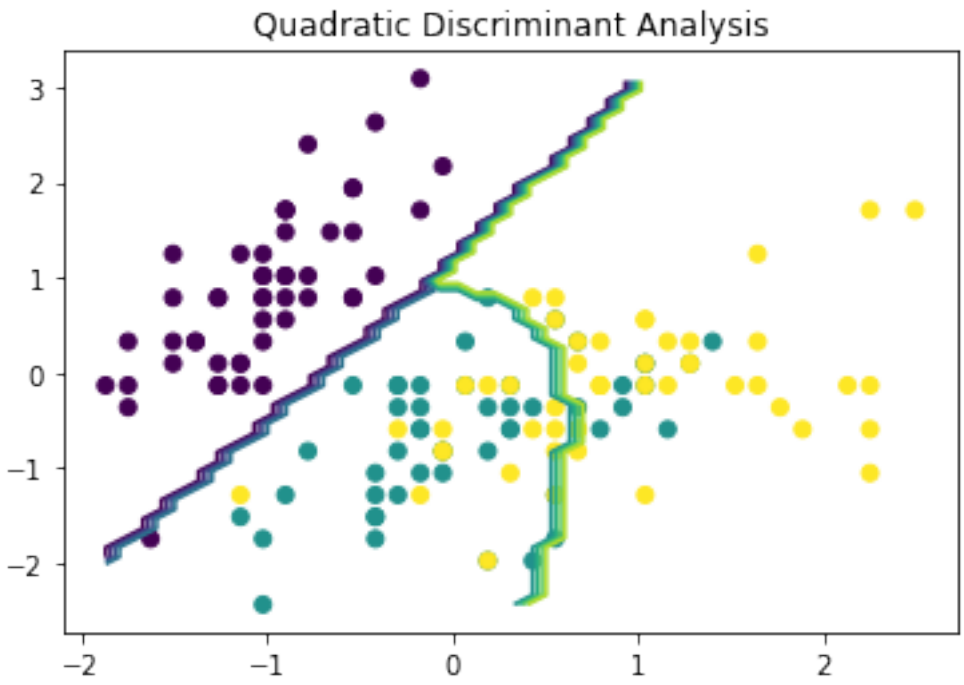
[176]:
```
# Figure (frontière de décision)
x_min, x_max, y_min, y_max = np.min(X[:, 0]),np.max(X[:, 0]) ,  np.min(X[:,␣
 ↪1]),  np.max(X[:, 1])
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.05),np.arange(y_min, y_max, 0.
 ↪05))
Z = lda.fit(X, y).predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.show()
plt.scatter(X[:,0],X[:,1], c=y)
plt.contour(xx, yy, Z)
plt.title("Linear Discriminant Analysis")
plt.show()
```



[177]:
```
#-- QDA
qda = QuadraticDiscriminantAnalysis()
qda.fit(X, y)
y_pred = qda.predict(X)
print ("Taux d'erreur QDA", np.mean(y_pred != y))
```

```
Taux d'erreur QDA 0.2
```

[178]:
```
# Figure (frontière de décision)
x_min, x_max, y_min, y_max = np.min(X[:, 0]),np.max(X[:, 0]) ,  np.min(X[:,␣
 ↪1]),  np.max(X[:, 1])
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),np.arange(y_min, y_max, 0.1))
Z = qda.fit(X, y).predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.show()
plt.scatter(X[:,0],X[:,1], c=y)
plt.contour(xx, yy, Z)
plt.title("Quadratic Discriminant Analysis")
plt.show()
```



1.3.2 2. Votre version des algorithmes LDA et QDA

Astuce : * `np.cov` calcul de covariance d'une numpy array (`bias=True`pour obtenir des résultats identiques à l'algorithme QDA de Scikit-Learn) * loi normale multivariée : $p(z) = \frac{1}{(2\pi)^{\frac{d}{2}}|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(z-\mu)^T\Sigma^{-1}(z-\mu)}$ * `np.linalg.det` et `np.linalg.inv` calcul de déterminant et de l'inverse d'une numpy array * `np.einsum` (non obligatoire)

[211]:
```
def covariance(x,bias=False):
    return np.cov(np.transpose(x),bias=bias)
```

[212]:
```
#-- Test
cov = covariance(X)
print(cov)
print(cov.shape)
```

```
[[ 1.00671141 -0.11835884]
 [-0.11835884  1.00671141]]
(2, 2)
```

Résultat attendu [[1.00671141 -0.11010327] [-0.11010327 1.00671141]] (2, 2)

[378]:
```
#-- loi normale multivariée
def loi_normale_multivarie(X,mu,Sigma):
    m = X.shape[0]
```

```
    ret = np.zeros(m)
    for i in range(m):
        sub = X[i] - mu
        f = np.dot(np.dot(sub,np.linalg.inv(Sigma)),np.transpose(sub))
        ret[i] = np.exp(-f/2) / (np.sqrt((2*np.pi)**X.shape[1]*np.linalg.
↪det(Sigma)))
    return ret
```

[379]:
```
print(loi_normale_multivarie(X,[0, 0],[[1, 0], [0,1]])[:10])
```

```
[0.06312267 0.08209894 0.05776305 0.05091985 0.04327343 0.02099329
 0.037486   0.06917792 0.03229923 0.08241859]
```

Résultat attendu [0.06312267 0.08209894 0.05776305 0.05091985 0.04327343 0.02099329 0.037486 0.06917792 0.03229923 0.08241859]

LDA

[389]:
```
def estimation_param_lda(Xtrain, ytrain):
    nbClasses = proba_classe(ytrain).shape[0]
    mu = []
    for i in range(nbClasses):
        classe =[]
        for j in range(ytrain.shape[0]):
            if ytrain[j] == i:
                classe.append(Xtrain[j])
        mu.append(np.mean(classe,axis=0))
    return (np.asarray(mu),covariance(Xtrain,bias=True))
```

[390]:
```
print(estimation_param_lda(X,y))
```

```
(array([[-1.01457897,  0.85326268],
       [ 0.11228223, -0.66143204],
       [ 0.90229674, -0.19183064]]), array([[ 1.        , -0.11756978],
       [-0.11756978,  1.        ]]))
```

Résultat attendu (la valeur de Sigma peut différer en fonction du calcul de variance réalisé) (array([[-1.01457897, 0.85326268], [0.11228223, -0.66143204], [0.90229674, -0.19183064]]), array([[1. , -0.11756978], [-0.11756978, 1.]]))

[398]:
```
#-- vraisemblance (Pr(X=x|Y=y)) en fonction des paramètres mu et sigma
def vraisemblance_lda(X, mu, Sigma):
    N, d = X.shape
    C = mu.shape[0]
    pvrais = np.zeros((N,C))
    for c in range(C):
        norm = loi_normale_multivarie(X,mu[c],Sigma)
        for dim in range(N):
            pvrais[dim][c] = np.prod(norm[dim])
    return pvrais
```

[392]:
```
#-- LDA
def predict_lda(X, Xtrain, ytrain):
    mu, Sigma = estimation_param_lda(Xtrain,ytrain)
    yhat = np.argmax(proba_classe(ytrain) * vraisemblance_lda(X, mu, Sigma),␣
 ↪axis=1)
    return yhat
```

[399]:
```
#-- test
yhat = predict_lda(X,X,y)
print ("Taux d'erreur LDA", 1-getOA(y,yhat))
print ("\tet coefficient Kappa", getKappa(y,yhat))
```

```
Taux d'erreur LDA 0.21333333333333337
        et coefficient Kappa 0.6799999999999999
```

Résultat attendu Taux d'erreur LDA 0.21333333333333337 et coefficient Kappa 0.6799999999999999

QDA

[401]:
```
def estimation_param_qda(Xtrain, ytrain):
    nbClasses = proba_classe(ytrain).shape[0]
    mu = []
    Sigma = []
    for i in range(nbClasses):
        classe =[]
        for j in range(ytrain.shape[0]):
            if ytrain[j] == i:
                classe.append(Xtrain[j])
        Sigma.append(covariance(classe,bias=True))
        mu.append(np.mean(classe,axis=0))
    return (np.asarray(mu),np.asarray(Sigma))
```

[402]:
```
#-- vraisemblance (Pr(X=x|Y=y)) en fonction des paramètres mu et sigma
def vraisemblance_qda(X, mu, Sigma):
    N, d = X.shape
    C = mu.shape[0]
    pvrais = np.zeros((N,C))
    for c in range(C):
        norm = loi_normale_multivarie(X,mu[c],Sigma[c])
        for dim in range(N):
            pvrais[dim][c] = np.prod(norm[dim])
    return pvrais
```

[403]:
```
#-- QDA
def predict_qda(X, Xtrain, ytrain):
    mu, Sigma = estimation_param_qda(Xtrain,ytrain)
    yhat = np.argmax(proba_classe(ytrain) * vraisemblance_qda(X, mu, Sigma),␣
 ↪axis=1)
```

```
    return yhat
```

[404]:
```
#-- test
yhat = predict_qda(X,X,y)
print ("Taux d'erreur QDA", 1-getOA(y,yhat))
print ("\tet coefficient Kappa", getKappa(y,yhat))
```

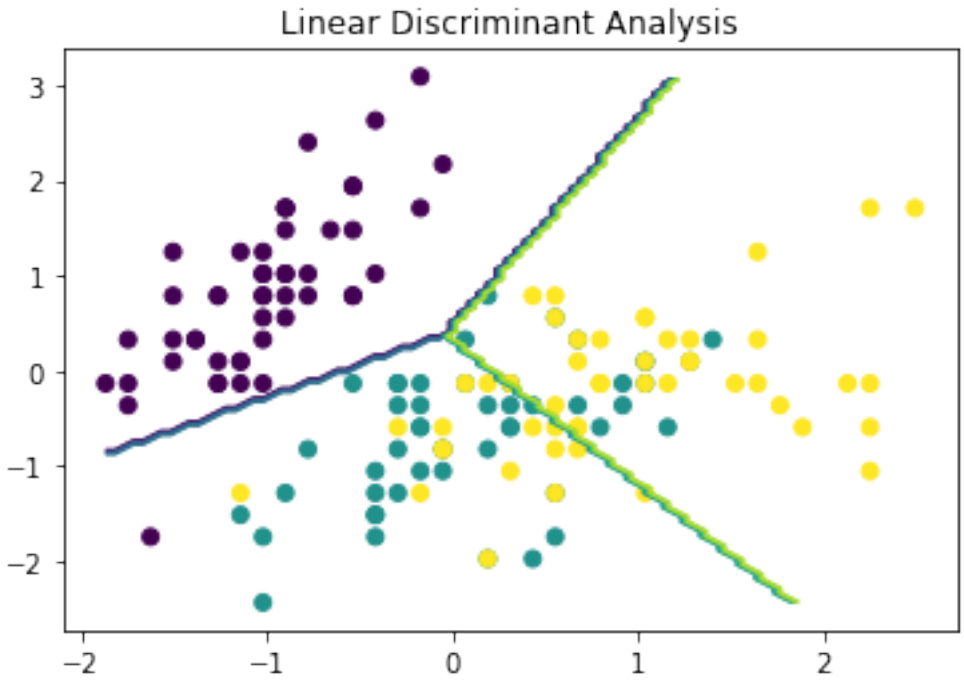
```
Taux d'erreur QDA 0.19999999999999996
        et coefficient Kappa 0.7000000000000001
```

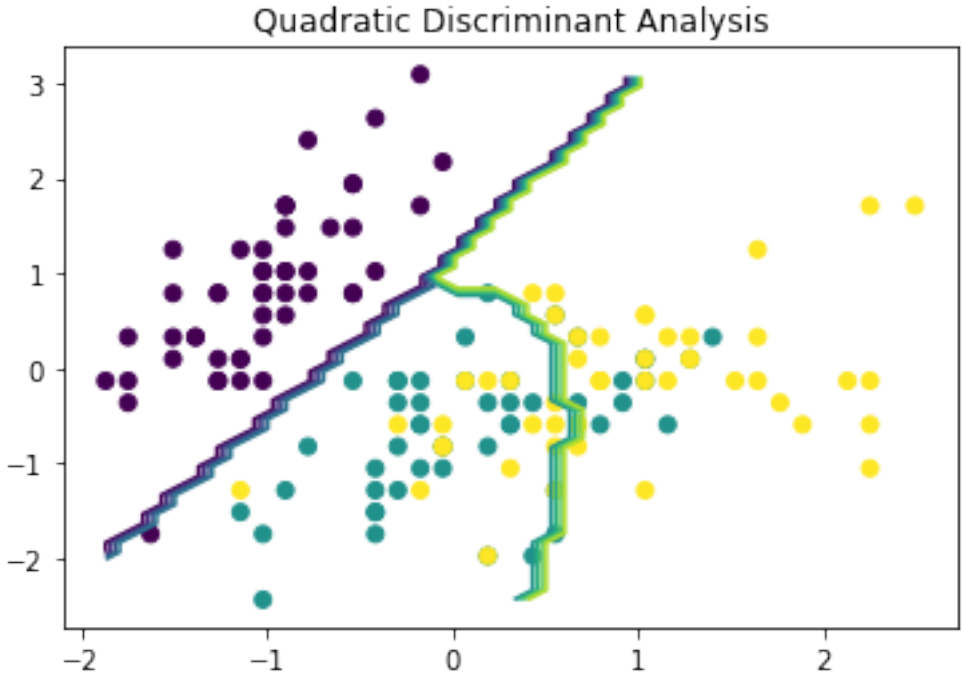
Résultat attendu Taux d'erreur QDA 0.19999999999999996 et coefficient Kappa 0.7000000000000001 **Note** : On remarque que OA(LDA) > OA(QDA), mais Kappa(LDA)<Kappa(QDA).

Oui.

1.3.3 3. Visualisation des frontières de décision

[405]:
```
# Figure (frontière de décision)
x_min, x_max, y_min, y_max = np.min(X[:, 0]),np.max(X[:, 0]) ,  np.min(X[:,␣
 ↪1]),  np.max(X[:, 1])
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.05),np.arange(y_min, y_max, 0.
 ↪05))
Z = predict_lda(np.c_[xx.ravel(), yy.ravel()],X,y)
Z = Z.reshape(xx.shape)

plt.show()
plt.scatter(X[:,0],X[:,1], c=y)
plt.contour(xx, yy, Z)
plt.title("Linear Discriminant Analysis")
plt.show()

# Figure (frontière de décision)
x_min, x_max, y_min, y_max = np.min(X[:, 0]),np.max(X[:, 0]) ,  np.min(X[:,␣
 ↪1]),  np.max(X[:, 1])
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),np.arange(y_min, y_max, 0.1))
Z = predict_qda(np.c_[xx.ravel(), yy.ravel()],X,y)
Z = Z.reshape(xx.shape)

plt.show()
plt.scatter(X[:,0],X[:,1], c=y)
plt.contour(xx, yy, Z)
plt.title("Quadratic Discriminant Analysis")
plt.show()
```





Note : Il est possible que le résultat visuel pour votre implémentation de LDA diffère de celle de Scikit-Learn vu précédemment. Cette différence est due à l'utilisation d'un algorithme d'optimisation pour l'apprentissage des paramètres du modèle.

Les plots ne diffère pas car le bias est à True.

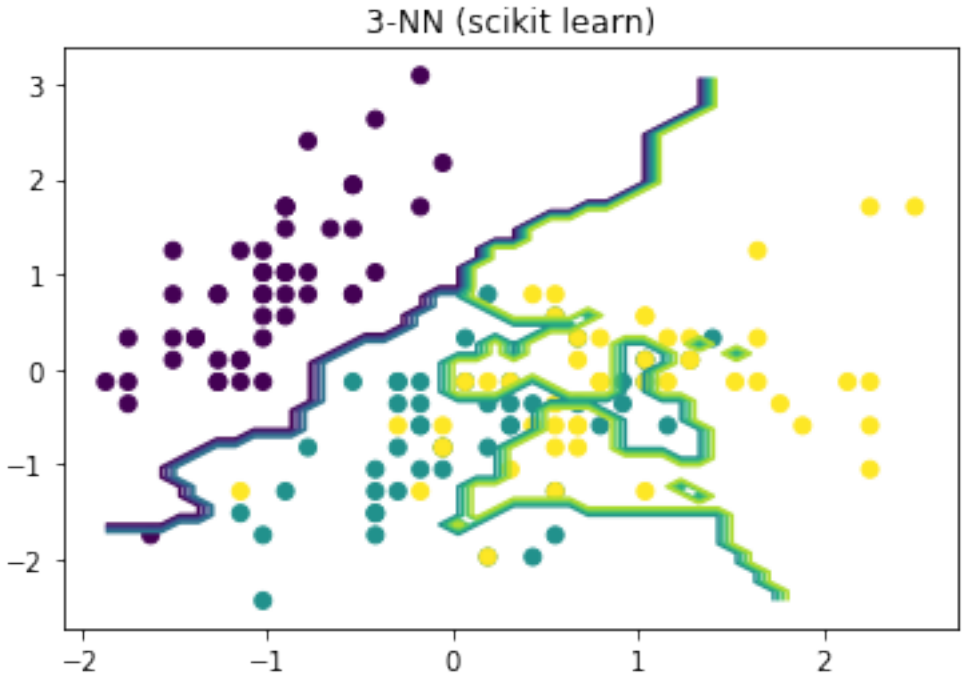
1.4 *k*-Plus Proches Voisins

1.4.1 1. Scikit-learn

[406]:
```
#-- k-PPV
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X, y)
y_pred = knn.predict(X)
print ("Taux d'erreur du 3-PPV", np.mean(y_pred != y))
```

```
Taux d'erreur du 3-PPV 0.14666666666666667
```

[407]:
```
# Figure
x_min, x_max, y_min, y_max = np.min(X[:, 0]),np.max(X[:, 0]) ,  np.min(X[:,␣
↪1]),  np.max(X[:, 1])
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),np.arange(y_min, y_max, 0.1))
Z = knn.fit(X, y).predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.show()
plt.scatter(X[:,0],X[:,1], c=y)
plt.contour(xx, yy, Z)
plt.title("3-NN (scikit learn)")
plt.show()
```



1.4.2 2. Votre version de l'algorithme des k-Plus Proches Voisins

Astuces : * `euclidean_distances()` calculer les distances entre échantillons (doc) * `np.argsort()` donner les indices des points * `np.unique()` pour avoir les éléments uniques d'une liste (et leur compte `return_counts=True`) * `map(lambda i:y[i], tri.ravel())` donner les valeurs de y associées au i-ième indice, stocké dans la liste appelée tri

[534]:
```
#-- Votre algorithme
def kPPV(X, Xtrain, ytrain, k=5):
    m = X.shape[0]
    ret = np.zeros(m,dtype=int)
    for i in range(m):
        sort = np.argsort(euclidean_distances(Xtrain,[X[i]]).flatten())
        classe = []
        w = k
        for j in sort[:k]:
            classe += [ytrain[j]] * w #pondération
            w -= 1
        uni, count = np.unique(classe,return_counts=True)
        ret[i] = uni[np.argmax(count)]
    return ret
```

[532]:
```
kPPV(np.array([[0.5, 1.3],[-3.5,0.4],[1.5,-0.4]]),X,y, k=5)
```

[532]:
```
array([2, 0, 2])
```

Résultat attendu array([2, 0, 2])

[533]:
```
#-- test
for k in [1,3,5,10,15,30]:
    yhat = kPPV(X,X,y,k=k)
    print ("Taux d'erreur "+str(k)+"-PPV", 1-getOA(y,yhat))
    print ("\tet coefficient Kappa", getKappa(y,yhat))
```

```
Taux d'erreur 1-PPV 0.07333333333333336
        et coefficient Kappa 0.8899999999999998
Taux d'erreur 3-PPV 0.10666666666666669
        et coefficient Kappa 0.84
Taux d'erreur 5-PPV 0.14
        et coefficient Kappa 0.7899999999999998
Taux d'erreur 10-PPV 0.1466666666666666
        et coefficient Kappa 0.7799999999999999
Taux d'erreur 15-PPV 0.15333333333333332
        et coefficient Kappa 0.77
Taux d'erreur 30-PPV 0.17333333333333334
        et coefficient Kappa 0.74
```

Résultat attendu Taux d'erreur 1-PPV 0.06000000000000005 et coefficient Kappa

0.9099999999999999 Taux d'erreur 3-PPV 0.15333333333333332 et coefficient Kappa 0.77 Taux d'erreur 5-PPV 0.16000000000000003 et coefficient Kappa 0.7599999999999999 Taux d'erreur 10-PPV 0.15333333333333332 et coefficient Kappa 0.77 Taux d'erreur 15-PPV 0.18666666666666665 et coefficient Kappa 0.72 Taux d'erreur 30-PPV 0.18000000000000005 et coefficient Kappa 0.7299999999999999 **Note #1** : Le taux d'erreur du 1-PPV devrait être de 0 % sur les échantillons d'apprentissage (chaque donnée d'apprentissage est la plus proche d'elle-même). **Comment pouvez vous expliquer le taux d'erreur de 6 % observé ?**

Note #2 : Il est rare d'utiliser une implémentation naïve du k-PPV qui calcule une distance entre chaque paire d'échantillons (coûteuse en mémoire et en temps). Pour aller plus loin : K-d tree.

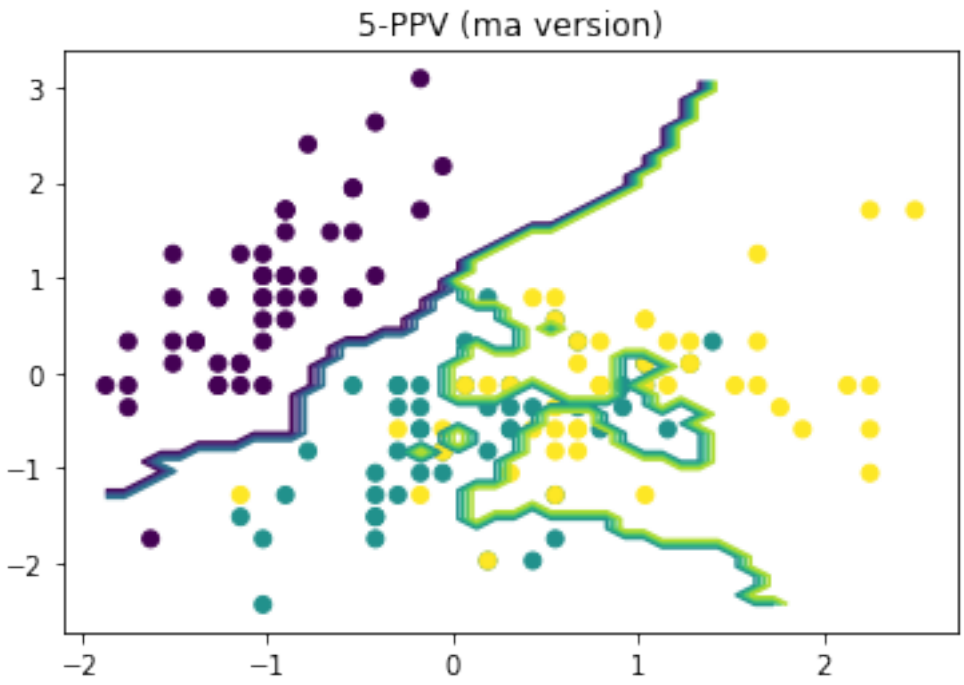
Note #3 : En plus de l'hyperparamètre k à fixer, le choix de la distance est aussi un hyperparamètre. Dans cet exemple, nous utilisons la distance euclidienne, mais vous pouvez tester d'autres types de distance (par exemple Minkowski pour $p \neq 2$)

Ce taux d'erreur est sûrement dû au fait du tirage aléatoire lorqu'on à des distances égales lors du choix de la classe.

On remarque qu'avec une pondération des classes en fonction de leur distance, le coefficient Kappa est moins bon pour des k petits, et s'améliore avec des k plus grand.

1.4.3 3. Visualisation des frontières de décision

[536]:
```
# Figure
x_min, x_max, y_min, y_max = np.min(X[:, 0]),np.max(X[:, 0]) ,  np.min(X[:,␣
↪1]),  np.max(X[:, 1])
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),np.arange(y_min, y_max, 0.1))
Z = kPPV(np.c_[xx.ravel(), yy.ravel()],X,y,k=5)
Z = Z.reshape(xx.shape)

plt.show()
plt.scatter(X[:,0],X[:,1], c=y)
plt.contour(xx, yy, Z)
plt.title("5-PPV (ma version)")
plt.show()
```


5-PPV (ma version)
3
2
1
0
−1
−2
−2
−1
0
1
2

TP8

May 11, 2020

1 TP08 - Sélection de modèles

L'objectif de ce TP est de mettre en oeuvre une * de mesurer le risque empirique d'un modèle d'apprentissage supervisé pour différentes tailles d'échantillon (*i.e.*, différentes quantités d'échantillons d'apprentissage) * de sélectionner la valeur optimale et d'évaluer la capacité de généralisation d'un modèle d'apprentissage supervisé (classification) en utilisant * un découpage en données d'apprentissage, de validation et de test * le principe de la validation croisée

[1]:
```
import numpy as np
import pandas as pd

import random

import seaborn as sn

import matplotlib.pyplot as plt
%matplotlib inline

from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from numpy.random import randint
```

1.1 Partie 1. Les données

Les données utilisées dans le cadre de ce TP sont des images de chiffres écrit manuscritement. La reconnaissance automatique de ces chiffres (et plus généralement d'écriture manuscrite) est un problème classique mais complexe. Elle est notamment utilisée dans les systsèmes de tri automatique du courrier. Les données utilisées pour ce TP correspondent à un extrait simplifié des données MNIST (Modified National Institure of Standards and Technology) qui contient à l'origine 70 000 images de chiffres de 0 à 9.

Chaque image (*i.e.*, une observation) a une taille de 8 pix. $\times$ 8 pix. Ce qui corresponds à un vecteur composées de $d = 64$ variables ($\mathbf{x}_i \in \mathbb{R}^d$ pour $1 \leq i \leq m$).

Note : Les imagettes de taille de 8 pix. $\times$ 8 pix sont obtenues en applicant une réduction de dimension aux images binaires originales de taille 32 pix. $\times$ 32 pix. Les images originales ont été découpées en blocs de 4 pix. $\times$ 4 pix. (sans chevauchement). Dans chaque bloc, on a compté le

nombre de “1” qui sert de nouvelles valeurs de pixel pour créer une image (non-binaire) de taille 8 pix. $\times$ 8 pix avec chaque pixel pouvant avoir une valeur comprise entre 0 et 16.

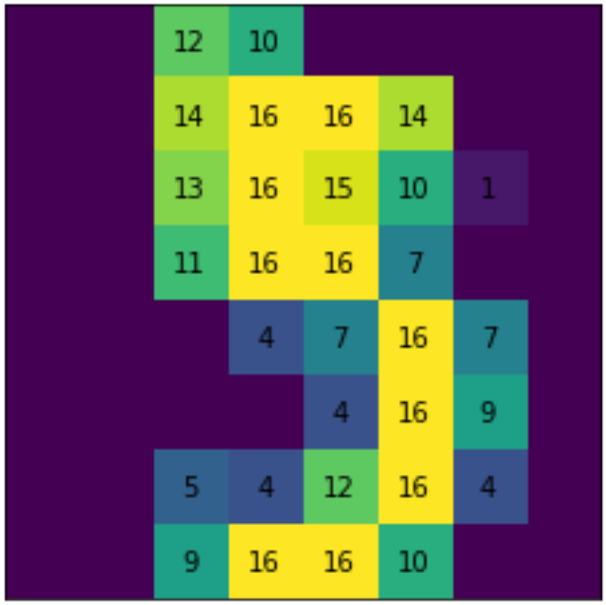
[2]:
```
#-- Chargement des données
digits = datasets.load_digits()
X = digits.data
Y = digits.target
print(digits.DESCR)
```

```
.. _digits_dataset:

Optical recognition of handwritten digits dataset
--------------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 5620
    :Number of Attributes: 64
    :Attribute Information: 8x8 image of integer pixels in the range 0..16.
    :Missing Attribute Values: None
    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
    :Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets
https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digit
s

The data set contains images of hand-written digits: 10 classes where
each class refers to a digit.

Preprocessing programs made available by NIST were used to extract
normalized bitmaps of handwritten digits from a preprinted form. From a
total of 43 people, 30 contributed to the training set and different 13
to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of
4x4 and the number of on pixels are counted in each block. This generates
an input matrix of 8x8 where each element is an integer in the range
0..16. This reduces dimensionality and gives invariance to small
distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G.
T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.
L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,
1994.

.. topic:: References

  - C. Kaynak (1995) Methods of Combining Multiple Classifiers and Their
    Applications to Handwritten Digit Recognition, MSc Thesis, Institute of
```

Graduate Studies in Science and Engineering, Bogazici University.
- E. Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.
- Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin. Linear dimensionalityreduction using relevance weighted LDA. School of Electrical and Electronic Engineering Nanyang Technological University. 2005.
- Claudio Gentile. A New Approximate Maximal Margin Classification Algorithm. NIPS. 2000.

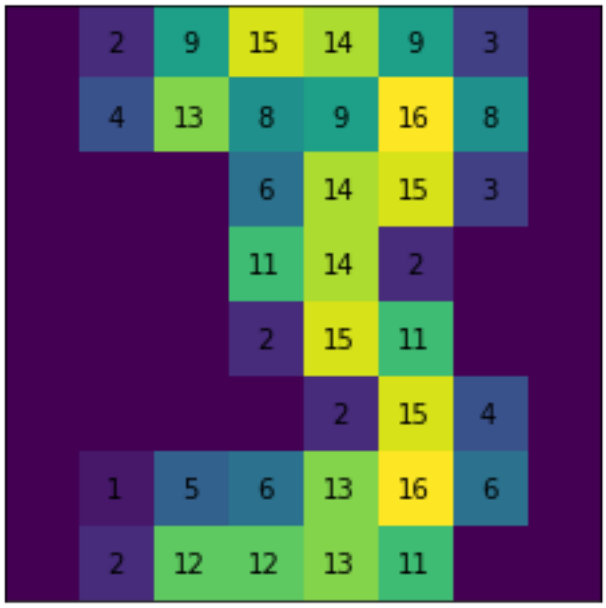
[3]:
```
#-- Visualisation de données
disp_imgno = [5, 13] # Tester d'autres valeurs
for ll in disp_imgno:
    plt.figure()
    plt.matshow(digits.images[ll])
    for (i, j), z in np.ndenumerate(digits.images[ll]):
        if z!=0:
            plt.text(j, i, '{:d}'.format(int(z)), ha='center', va='center')
    plt.gca().set_xticks([])
    plt.gca().set_yticks([])
    plt.show()
    print("Variables: ", X[ll,:])
    print("Classe asociée: ", Y[ll])
```

```
<Figure size 432x288 with 0 Axes>
```



```
Variables:  [ 0.  0. 12. 10.  0.  0.  0.  0.  0.  0. 14. 16. 16. 14.  0.  0.  0.
0.
 13. 16. 15. 10.  1.  0.  0.  0. 11. 16. 16.  7.  0.  0.  0.  0.  0.  4.
  7. 16.  7.  0.  0.  0.  0.  0.  4. 16.  9.  0.  0.  0.  5.  4. 12. 16.
  4.  0.  0.  0.  9. 16. 16. 10.  0.  0.]
Classe asociée:  5
```

```
<Figure size 432x288 with 0 Axes>
```




```
Variables:  [ 0.  2.  9. 15. 14.  9.  3.  0.  0.  4. 13.  8.  9. 16.  8.  0.  0.
0.
  0.  6. 14. 15.  3.  0.  0.  0.  0. 11. 14.  2.  0.  0.  0.  0.  0.  2.
 15. 11.  0.  0.  0.  0.  0.  0.  2. 15.  4.  0.  0.  1.  5.  6. 13. 16.
  6.  0.  0.  2. 12. 12. 13. 11.  0.  0.]
Classe asociée:  3
```

Questions : * Quel est le lien entre les images (`digit.images[ll]`) et les variables (`X[ll,:]`) + la classe associée (`Y[ll]`)? * Quel est le nombre de données d'apprentissage ?

X[ll,:] et digit.images[ll] correspond à l'image à la position ll qui sont respectivement non dimensionnée et dimensionnée. Y[ll] correspond à la classe associée à cette image. Si le résultat est n, alors on peut dire que l'image représente le chiffre n.

Le nombre de données d'apprentissage est la taille de digit.images : 1797.

1.2 Partie 2. Influence de la taille de l'échantillon

L'objectif ici est d'évaluer les performances d'un algorithme de classification (risque empirique et risque réel), le k-Plus Proches Voisins (PPV), lorsque le nombre de données d'apprentissage augmente (m).

Question : Rappelez brièvement le principe de fonctionnement du k-PPV. Principe de fonctionnement du k-PPV:

Pour un échantillon donné, on regarde ses k plus proches voisins afin de lui attribuer la classe qui est majoritaire. On répète ce procédé pour tout les échantillons.

1.2.1 1. Risque empirique

2.1.a Appliquer un k-PPV avec $k = 1$. Calculer le taux d'erreur (qui corresponds au risque empirique $\mathcal{R}_{emp}$) et expliquer le résultat.

Astuces : * Vous pouvez utiliser l'algorithme des k-PPV de Scikit-Learn: doc. Trouvez le nom de l'hyperparamètre qui vous permet de régler le nombre de voisins considéré lors de l'apprentissage (k). * Le taux d'erreur est égal à 1 - Overall Accuracy (vous pouvez utiliser votre implémentation du TP précédent).

[4]:
```
def proba_classe(y):
    return np.bincount(y) / y.shape[0]

def getConfusionMatrix(y,yhat,C=None):
    if C is None:
        C = proba_classe(y).shape[0]
    confMatrix = np.zeros((C,C))
    for i in range(y.shape[0]):
        confMatrix[y[i]][yhat[i]] += 1
    return confMatrix

def getOA(y,yhat, C=None):
    if C is None:
        C = getConfusionMatrix(y,yhat)
    return np.sum(C.diagonal()) / y.shape[0]

def sample(X,Y,size):
    x = []
    y = []
    for i in range(size):
        r = randint(X.shape[0])
        x.append(X[r,:])
        y.append(Y[r])
    x = np.asarray(x)
    y = np.asarray(y)
    return x,y
```

[5]:
```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X, Y)
print("Taux d'erreur :",1 - getOA(Y, knn.predict(X)))
```

```
Taux d'erreur : 0.0
```

Ce taux d'erreur de 0 s'explique par le fait qu'on initialise k à 1. En effet vu qu'on utilise les mêmes échantillons pour apprendre et pour tester, l'algorithme va à chaque fois regarder l'échantillon correspondant à lui-même et donc ne produira aucune erreur.

2.1.b Appliquer un k-PPV avec $k = 5$. Calculer le taux d'erreur $\mathcal{R}_{emp}$ et afficher la matrice de confusion.

Astuce : * Vous pouvez utiliser votre méthode `getConfusionMatrix()` développée au TP07, et comparer le résultat avec la matrice de confusion calculée par Scikit-Learn : `C = confusion_matrix(Y,Yhat)`. * Pour un affichage simple de la matrice de confusion :

```
plt.figure()
plt.imshow(C)
plt.show()
```

- Pour un affichage plus élaboré avec Seaborn :

```
labels = data.target_names
df_cm = pd.DataFrame(C, index = [i for i in labels],
                  columns = [i for i in labels])
plt.figure()
sn.heatmap(df_cm, annot=True, fmt='d')
```

[6]:
```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X, Y)
yhat = knn.predict(X)
c = getConfusionMatrix(Y,yhat)
print("Taux d'erreur :",1 - getOA(Y,yhat,C=c))

labels = digits.target_names.astype(np.int32)
df_cm = pd.DataFrame(c, index = [i for i in labels],
                  columns = [i for i in labels])
plt.figure()
sn.heatmap(df_cm, annot=True)
```

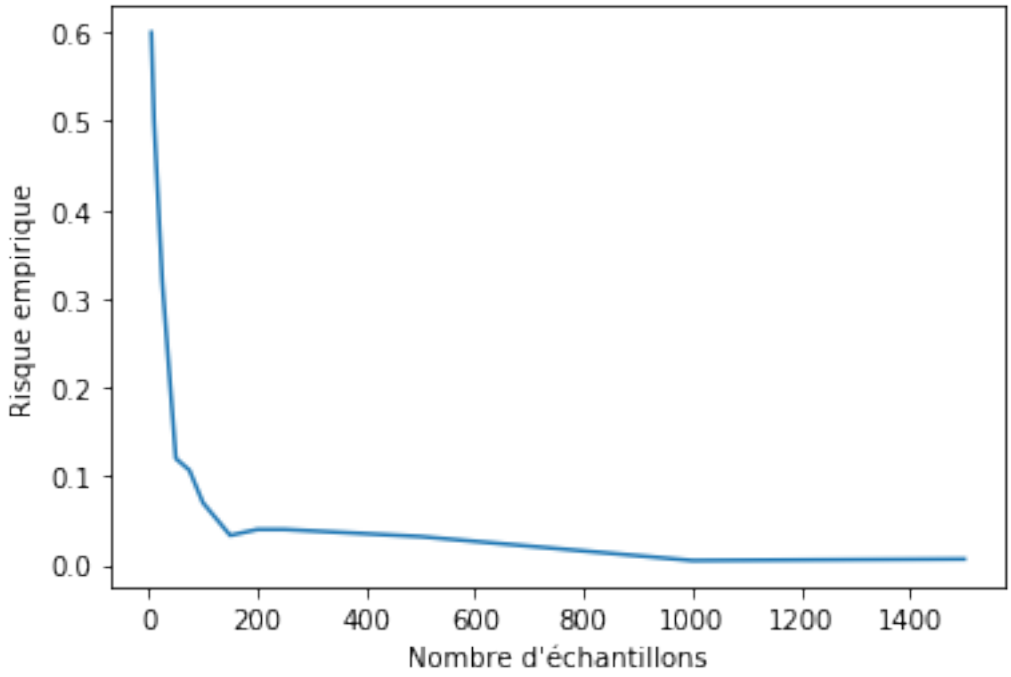
```
Taux d'erreur : 0.009460211463550361
```

[6]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x289b284fbc8>
```

2.1.c Tracer l'évolution du risque empririque, pour $k = 5$, en fonction de la taille des données d'apprentissage. * Définir les différentes tailles d'apprentissage : `size = [5, 10, 25, 50, 75, 100, 150, 200, 250, 500, 1000, 1500]`. * Tirer un échantillon aléatoire de la taille donnée. Note : il est possible d'utiliser la méthode `random.sample` du module `random`. * Applicquer un 5-PPV pour chaque taille d'échantillon et calculer $\mathcal{R}_{emp}$ le risque empirique. * Tracer l'évolution du risque empirique en fonction de la taille de l'échantillon : $\mathcal{R}_{emp} = f(size)$.

Commenter

[42]:
```
size = [5, 10, 25, 50, 75, 100, 150, 200, 250, 500, 1000, 1500]
res1 = []
knn = KNeighborsClassifier(n_neighbors=5)
for i in size:
    x, y = sample(X,Y,i)
    knn.fit(x, y)
    res1.append(1-getOA(y, knn.predict(x)))
plt.figure()
plt.plot(size,res1)
plt.xlabel("Nombre d'échantillons")
plt.ylabel("Risque empirique")
plt.show()
```



Commentaires:

Plus le nombre d'échantillons est élevé plus le risque empirique est faible.

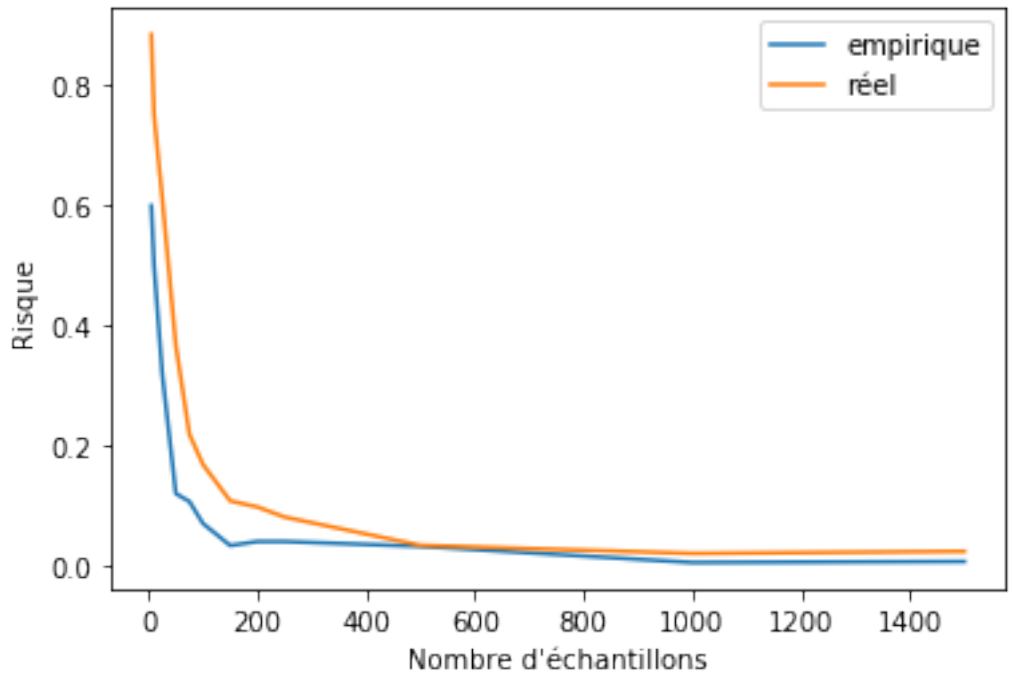
1.2.2 2. Risque réel et erreur de généralisation, *k* fixé

2.2.a Calculer une estimation de l'erreur de généralisation en utilisant un jeu d'apprentissage et de test ($N = 297$), pour les différentes tailles d'échantillons d'apprentissage testées ci-dessus (size), et toujours en considérant un 5-PPV. * Diviser aléatoirement les échantillons en sous-ensembles d'apprentissage et de test : `Xtrain, Xtest, Ytrain, Ytest = train_test_split(X,Y, test_size=N)` avec `N` la taille de l'échantillon de test désirée. * Superposer les deux courbes d'évolution du risque empirique et du risque réel en fonction de la taille de l'échantillon d'apprentissage. N'hésitez pas à aussi visualiser les courbes uniquement pour les plus grandes tailles d'échantillon d'apprentissage.

Comparer et commenter

[43]:
```
N = 297
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X,Y, test_size=N)
knn = KNeighborsClassifier(n_neighbors=5)
res2 = []
for i in size:
    x, y = sample(Xtrain,Ytrain,i)
    knn.fit(x, y)
    res2.append(1-getOA(Ytest, knn.predict(Xtest)))

plt.figure()
plt.plot(size,res1, label="empirique")
plt.plot(size,res2, label="réel")
```

```
plt.legend(loc="upper right")
plt.xlabel("Nombre d'échantillons")
plt.ylabel("Risque")
plt.show()
```



Comparaisons et commentaires

On remarque que le risque empirique est plus optimiste que le risque réel. La tendance de la courbe reste la même. **2.2.b** Jusqu'à présent les résultats sont obtenus pour une seule répétition alors que les résultats dépendent du découpage entre les données d'apprentissage et de test. Généralement cette procédure est répétée plusieurs fois afin d'obtenir la moyenne et la variance des résultats.

Répéter la procédure précédente (2.2.a) 20 fois, et calculer les risques empirique et réel moyens, ainsi que les variances associées. Comparer les résultats avec ceux obtenus sur une seule répétition.

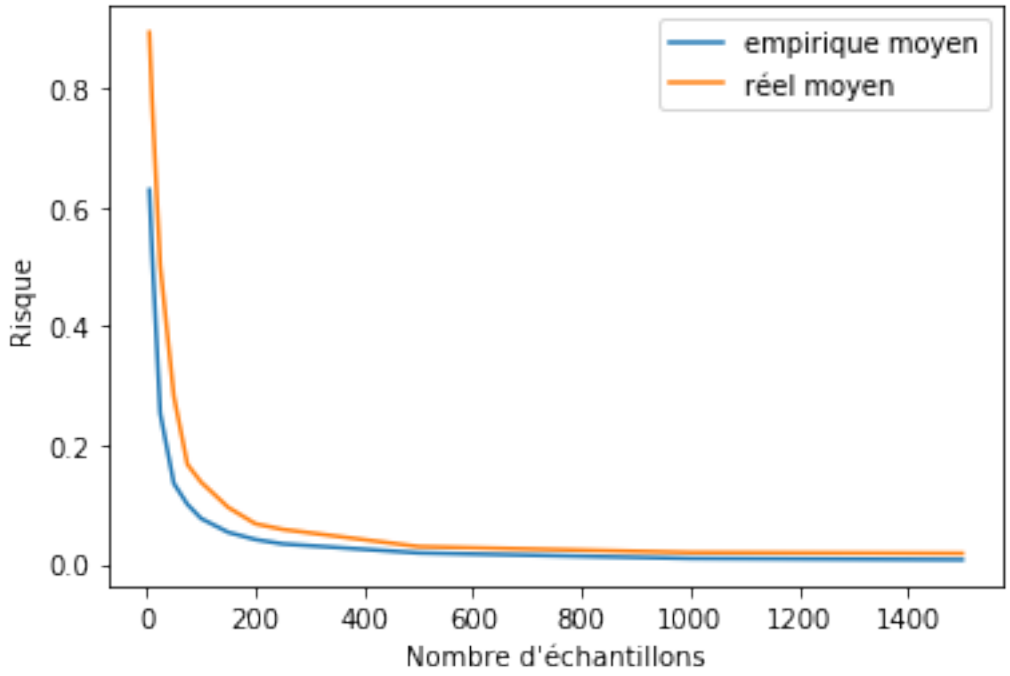
Astuce : vous pouvez afficher les résultats avec plt.errorbar() pour afficher la courbe +/- un écart-type

Commenter

[17]:
```
#-- Calculer pour 20 runs

rEmp = []
rReal = []
for i in range(20):
    resEmp = []
    resReal = []
    for j in size:
        x, y = sample(X,Y,j)
```

```
        knn.fit(x, y)
        resEmp.append(1-getOA(y, knn.predict(x)))
        x, y = sample(Xtrain,Ytrain,j)
        knn.fit(x, y)
        resReal.append(1-getOA(Ytest, knn.predict(Xtest)))
    rEmp.append(resEmp)
    rReal.append(resReal)
```

[28]:
```
#-- Affichage des résultats
plt.figure()
plt.plot(size,np.mean(rEmp, axis=0), label="empirique moyen")
plt.plot(size,np.mean(rReal, axis=0), label="réel moyen")
plt.legend(loc="upper right")
plt.xlabel("Nombre d'échantillons")
plt.ylabel("Risque")
plt.show()
```



Commentaires (comparaison avec le résultat précédent, comparaison de la variance, comparaison entre risque empirique et risque réel) Les courbes suivent la même tendance que les précédentes. Cependant elles sont plus lissent. Le risque réel reste au-dessus du risque empirique, c'est-à-dire que le risque empirique est plus optimiste que le risque réel. **2.2.c** Mettre en oeuvre une validation croisée sur 5 sous-ensembles / partitions (5-*fold cross validation*) sur l'ensemble des données d'apprentissage (toujours avec le 5-PPV). Donner une estimation de l'erreur de généralisation : `scores = cross_val_score(knn, X, Y, cv=5)` doc.

Question : Quelle est la valeur moyenne d'Overall Accuracy que vous obtenez ?

[33]:
```
scores = cross_val_score(knn, X, Y, cv=5)
print("Erreur :",1-np.mean(scores))
OA = []
for i in range(20):
    OA.append(getOA(Y,knn.predict(X)))
print("Moyenne OA :",np.mean(OA))
```

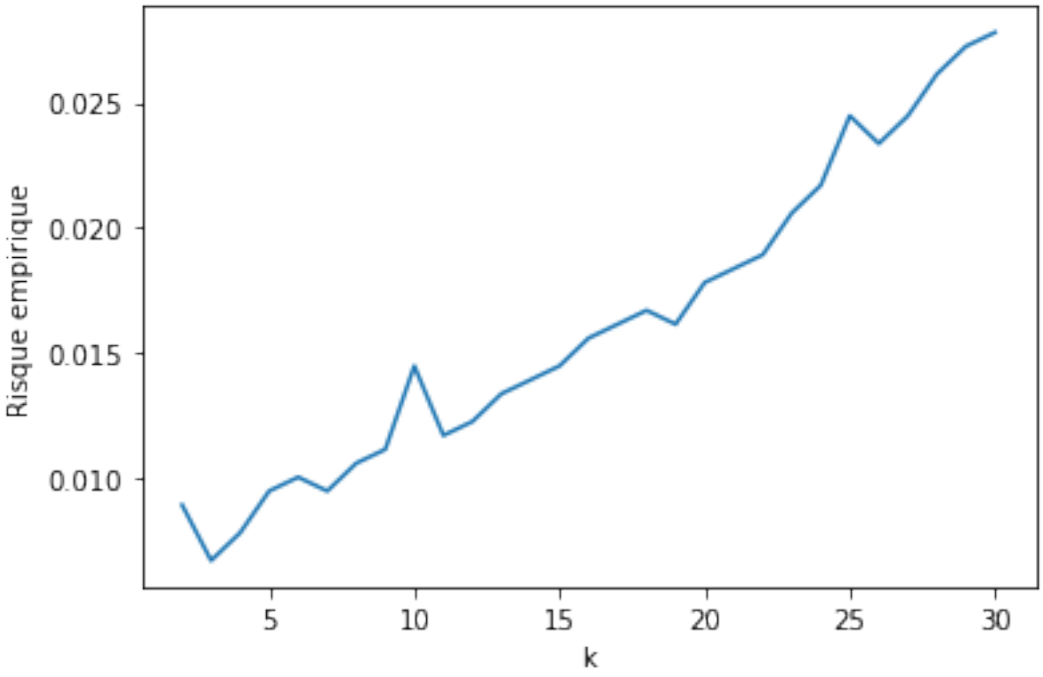
```
Erreur : 0.03727174249458387
Moyenne OA : 0.9816360601001671
```

1.3 Partie 3. Choix du meilleur hyperparamètre k (du k-PPV) et estimation des performances en généralisation

L'objectif de cette partie est de déterminer la meilleure valeur de l'hyperparamètre k de l'algorithme PPV et d'évaluer les performances de généralisation.

3.1 Tracer l'évolution du risque empirique pour $k \in [1, 30]$. **Commenter**

[66]:
```
rEmp = []
size = range(2,31)
for i in size:
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(X,Y)
    rEmp.append(1-getOA(Y, knn.predict(X)))

plt.figure()
plt.plot(size,rEmp)
plt.xlabel("k")
plt.ylabel("Risque empirique")
plt.show()
```



La valeur de k qui offre un risque empirique le moins élevé semble être 3

3.2 Créer un jeu d'apprentissage qui contient 80 % des données. Les 20 % restantes seront les données de test.

[44]:
```
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X,Y, test_size=0.2)
```

3.3 Pour $k \in [1, 30]$, réaliser une validation croisée sur 5 sous-ensembles (5-*fold cross-validation*) sur le jeu de données d'apprentissage. Le même jeu de donnée d'apprentissage sera considéré.

[58]:
```
err = 1
k = 0
size = range(2,31)
for i in size:
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(Xtrain,Ytrain)
    scores = cross_val_score(knn, Xtrain, Ytrain, cv=5)
    error = 1-np.mean(scores)
    if error <= err:
        err = error
        k  = i

print("K choisi :",k )
```

```
K choisi : 5
```

3.4 Quelle est la meilleure valeur de k à choisir ? Valeur de k optimale k = 5 **3.5** Pour cette valeur de k optimale, donner une estimation du risque réel. La phase d'apprentissage sera réalisée sur l'ensemble des données d'apprentissage.

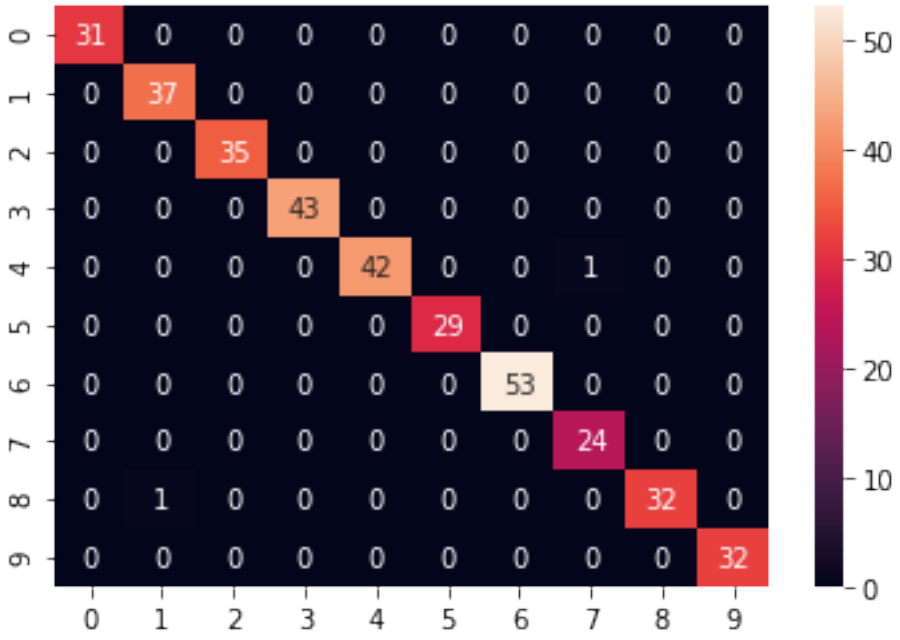
[61]:
```
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(Xtrain,Ytrain)
Yhat = knn.predict(Xtest)
print("Risque réel : ", 1-getOA(Ytest, Yhat))
```

```
Risque réel :  0.005555555555555536
```

3.6 Calculer la matrice de confusion du modèle. Que pensez vous de ce modèle ?

[62]:
```
C = confusion_matrix(Ytest,Yhat)

labels = digits.target_names.astype(np.int32)
df_cm = pd.DataFrame(C, index = [i for i in labels],
                     columns = [i for i in labels])
plt.figure()
sn.heatmap(df_cm, annot=True)
```

[62]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x289b2c8f408>
```



D'après les résultats, le modèle semble très fiable.

3.7 Finalement, donner une estimation du risque réel en répétant 10 fois l'expérience précédente.

Quelle est la valeur du risque réel moyen que vous obtenez ?

[64]:
```
rReal = []
for i in range(10):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(Xtrain,Ytrain)
```

```
    rReal.append(1-getOA(Ytest, knn.predict(Xtest)))
print("Valeur moyenne du risque réel :",np.mean(rReal))
```

```
Valeur moyenne du risque réel : 0.005555555555555536
```