HW1_DECHARRIN

October 25, 2023

de Charrin Théotime - TP1

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sn
import sklearn
import math
import scipy.stats as stats
from sklearn.preprocessing import Normalizer
from sklearn.model_selection import train_test_split
```

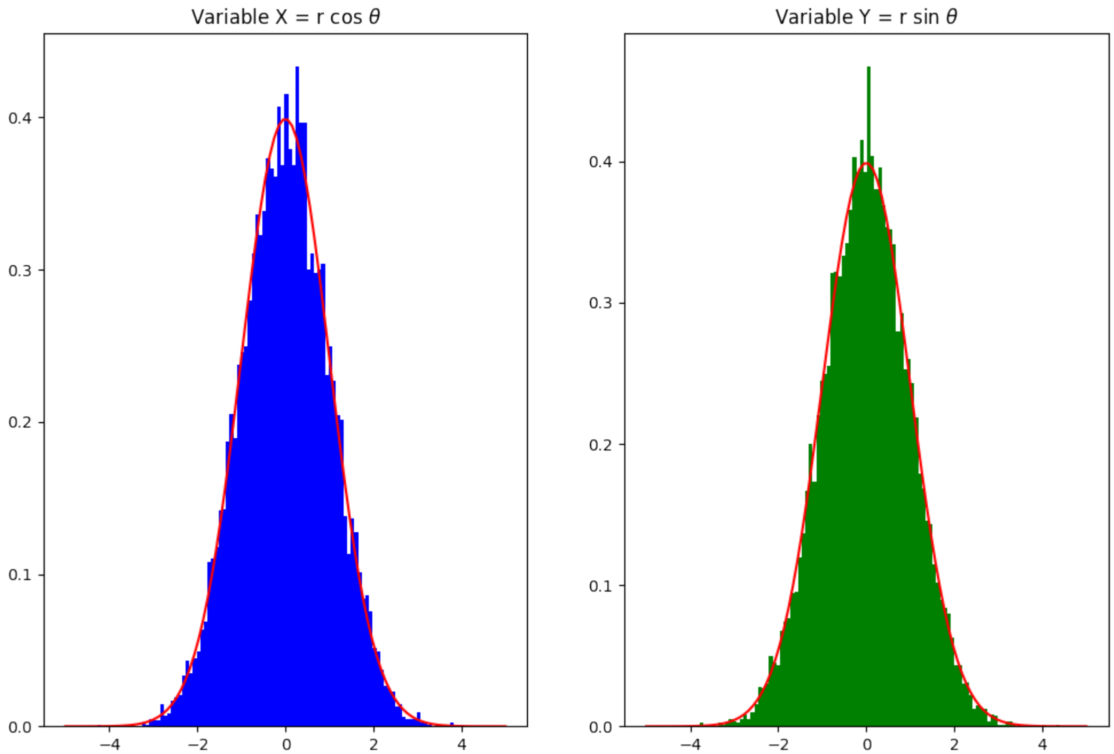
1 Exercice 1 - Implémentation des algorithmes de Box-Müller et Marsaglia-Bray

[2]:
```
def box_muller(n_sample):
    U=np.random.rand(n_sample,2)
    u=U[:,0]
    v=U[:,1]
    theta=2*np.pi*v
    r=np.sqrt(- 2*np.log(1-u))
    x=r*np.cos(theta)
    y=r*np.sin(theta)
    return x,y
```

[9]:
```
n=10000
x,y=box_muller(n)
ax1=plt.subplot(121)
ax2=plt.subplot(122)
ax1.hist(x,bins=int(n/100),density=True,color='b')
ax2.hist(y,bins=int(n/100),density=True,color='g')
ax1.set_title(r'Variable X = r cos $\theta$')
ax2.set_title(r'Variable Y = r sin $\theta$')

mu = 0
variance = 1
sigma = math.sqrt(variance)
```

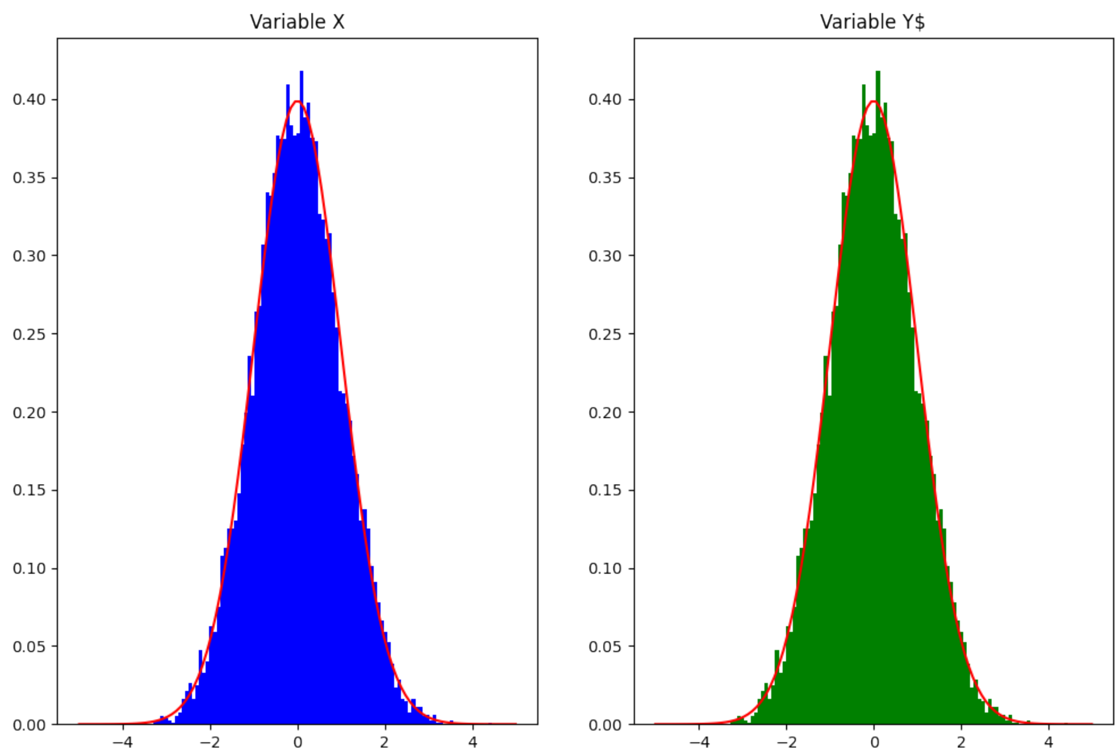
```
x = np.linspace(mu - 5*sigma, mu + 5*sigma, 100)
ax1.plot(x, stats.norm.pdf(x, mu, sigma),'r')
ax2.plot(x, stats.norm.pdf(x, mu, sigma),'r')
plt.show()
```



[7]:
```
def marsaglia_bray(n_sample):
    X=np.zeros(n_sample)
    Y=X
    for i in range(n_sample):
        V_1=np.random.uniform(-1,1)
        V_2=np.random.uniform(-1,1)
        while V_1**2+V_2**2 > 1:
            U_1=np.random.uniform(0,1)
            U_2=np.random.uniform(0,1)
            V_1=2*U_1-1
            V_2=2*U_2-1
        S=np.sqrt(-2*np.log(V_1**2+V_2**2))
        X[i]=S*(V_1/np.sqrt(V_1**2+V_2**2))
        Y[i]=S*(V_2/np.sqrt(V_1**2+V_2**2))
    return X,Y
```

[10]:
```
n=10000
x,y=marsaglia_bray(n)
ax1=plt.subplot(121)
ax2=plt.subplot(122)
ax1.hist(x,bins=int(n/100),density=True,color='b')
ax2.hist(y,bins=int(n/100),density=True,color='g')
ax1.set_title(r'Variable X')
ax2.set_title(r'Variable Y$')

mu = 0
variance = 1
sigma = math.sqrt(variance)
x = np.linspace(mu - 5*sigma, mu + 5*sigma, 100)
ax1.plot(x, stats.norm.pdf(x, mu, sigma),'r')
ax2.plot(x, stats.norm.pdf(x, mu, sigma),'r')
plt.show()
```



2 Exercice 3 - Stochastic Gradient learning in gardient descent

[11]:
```
plt.rcParams['figure.figsize'] = [12, 8]
plt.rcParams['figure.dpi'] = 100
```

2.0.1 Q1 - Describe the stochastic gradient descent algorithm for minimizing the empirical risk and implement it

La descente de gradient stochastique est une approximation de la descente de gradient classique, où l'on essaie de minimiser la fonction de risque. La seule différence est qu'au lieu de calculer le gradient "réel", on l'approxime grâce aux échantillons observés.

On va supposer :

$$\exists J \text{ différentiable, } \nabla J(\theta) = \mathbb{E}(g(\theta, X)), \text{ X de loi } \mathbb{P}_X \text{ connue et suivant } \ X^*$$

Notre étape de descente de gradient classique pour optimiser le paramètre θ donne ceci :

$$\begin{aligned} \theta^{k+1} &= \theta^k - \eta_{k+1} \nabla J(\theta^k) & (1) \\ &= \theta^k - \eta_{k+1} \mathbb{E}(g(\theta, X)) & (2) \end{aligned}$$

Avec η_k le Learning Rate à l'étape k. En descente de gradient stochastique, on va poser :

$$\mathbb{E}(g(\theta, X)) \approx \frac{1}{n} \sum_n g(\theta, X_k), X_k \text{ n copies iid de X}$$

Ainsi, si l'on fait des pas suffisamment petits, on peut poser :

$$\theta^{k+1} = \theta^k - \frac{c}{n} \sum_n g(\theta, X_{k+1})$$

Ici, on veut minimiser le risque sur l'ensemble des w, dans l'implémentation stochastique on introduit donc le risque empirique que l'on veut minimiser :

$$\min_w R_n(w) = \min_w \frac{1}{n} \sum_n (y_i - w^t x_i)^2$$

Le pseudo-algorithme est donc le suivant : > - On part d'un vecteur aléatoire $w_0 \in \mathbb{R}^d$ > - pour $k = (0, 1, \dots, \text{ fin } = n_{iter})$: > > - on choisit un learning rate $(\epsilon_k)_{k \geq 0} > 0$ et un $i \in \mathbb{N}$ aléatoire (MC) > > - On calcule $\nabla_w j(w^k, z_i)$ > > -

$$\begin{aligned} w_{k+1} &= w_k - \epsilon_k \nabla R_n(w_k) & (3) \\ &= w_k - \epsilon_k \mathbb{E}(\nabla_w j(w_k, z_i)) & (4) \\ &= w_k - \epsilon_k \nabla_w (y_i - w_k^t x_i)^2 & (5) \\ &= w_k + 2\epsilon_k (y_i - w_k^t x_i) x_i & (6) \end{aligned}$$

En pratique, on prendre un $(\epsilon_k)_{k \geq 0}$ tel que $\lim_{+\infty} \epsilon_k = 0$ mais pas trop vite, *i.e.* $\sum_{+\infty} \epsilon_k = +\infty$. On prendra souvent

$$\epsilon_k = \frac{1}{k^\alpha}, \ \alpha \in [0.1, 1]$$

[12]:
```
def stochastic_gd(x,y, w_0,alpha, niter):
    w_old=w_0
    k=0
    for k in range(1,niter+1):
        i=np.random.randint(0, x.shape[0])
        w_new=w_old+ (2./(k**alpha)) * (y[i] - x[i,:].dot(w_old))*x[i,:]
        w_old=w_new
        #On normalise le vecteur normal
    return w_new/np.sqrt(w_new.dot(w_new))
```

[13]:
```
def sample(n,w):
    x=np.random.rand(n,2)
    a=np.dot(x,w)
    a[a>0]=1
    a[a<0]=-1
    return x, a

w=np.array([1,-1])
#On normalise pour comparer les distances
w=w/np.sqrt(w.dot(w))
n=10000
x,y=sample(n,w)
print(x.shape)
```

```
(10000, 2)
```

[14]:
```
alpha=0.8
w_est=stochastic_gd(x,y,w,alpha,10000)
print(w_est)
y_est=x.dot(w_est)
y_est[y_est>0]=1
y_est[y_est<0]=-1
dist=np.linalg.norm(w_est-w)
print(f"La distance entre les deux vecteurs est de {dist}")
```

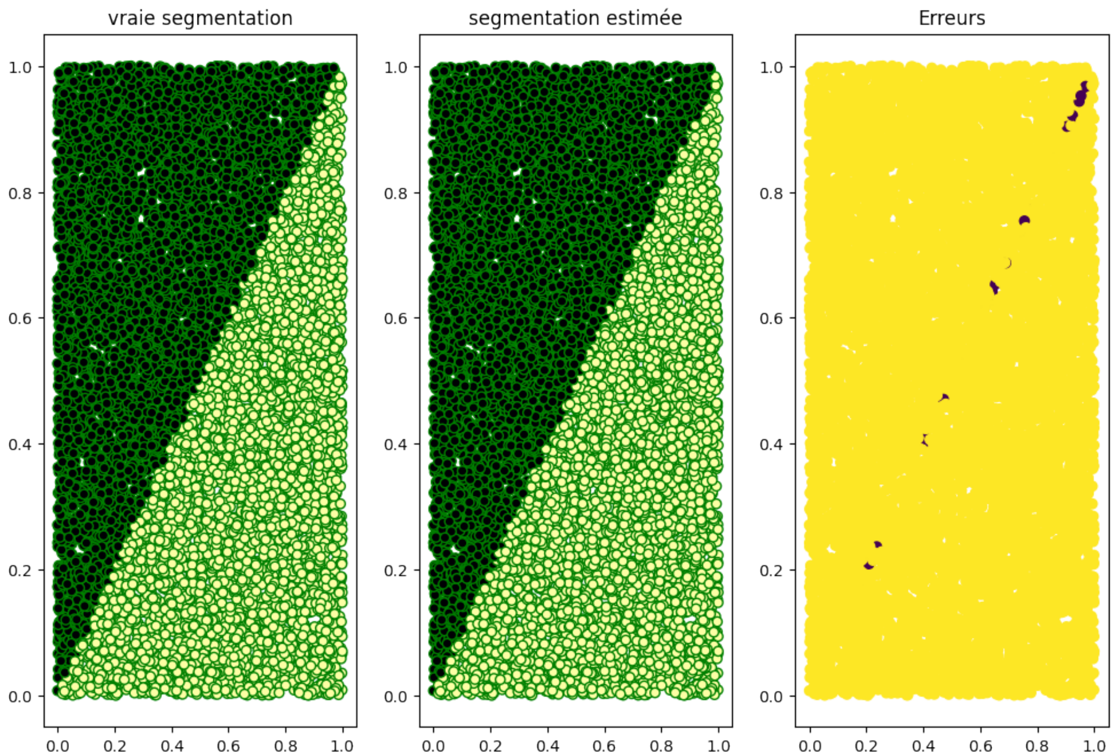
```
[ 0.70561589 -0.70859453]
La distance entre les deux vecteurs est de 0.0021062194486795873
```

Le vecteur estimé est toujours très proche de w*, mais jamais égal.

[16]:
```
ax1=plt.subplot(131)
ax1.scatter(x[:,0],x[:,1],c=y,cmap='inferno',facecolors='none',edgecolors='g')
ax1.set_title("vraie segmentation")
ax2=plt.subplot(132)
ax2.scatter(x[:,0],x[:
 ↪,1],c=y_est,cmap='inferno',facecolors='none',edgecolors='g')
ax2.set_title("segmentation estimée")
y_diff=y_est-y
```

```
ax3=plt.subplot(133)
ax3.scatter(x[:,0],x[:,1],c=y_diff)
ax3.set_title("Erreurs")
```

[16]: Text(0.5, 1.0, 'Erreurs')



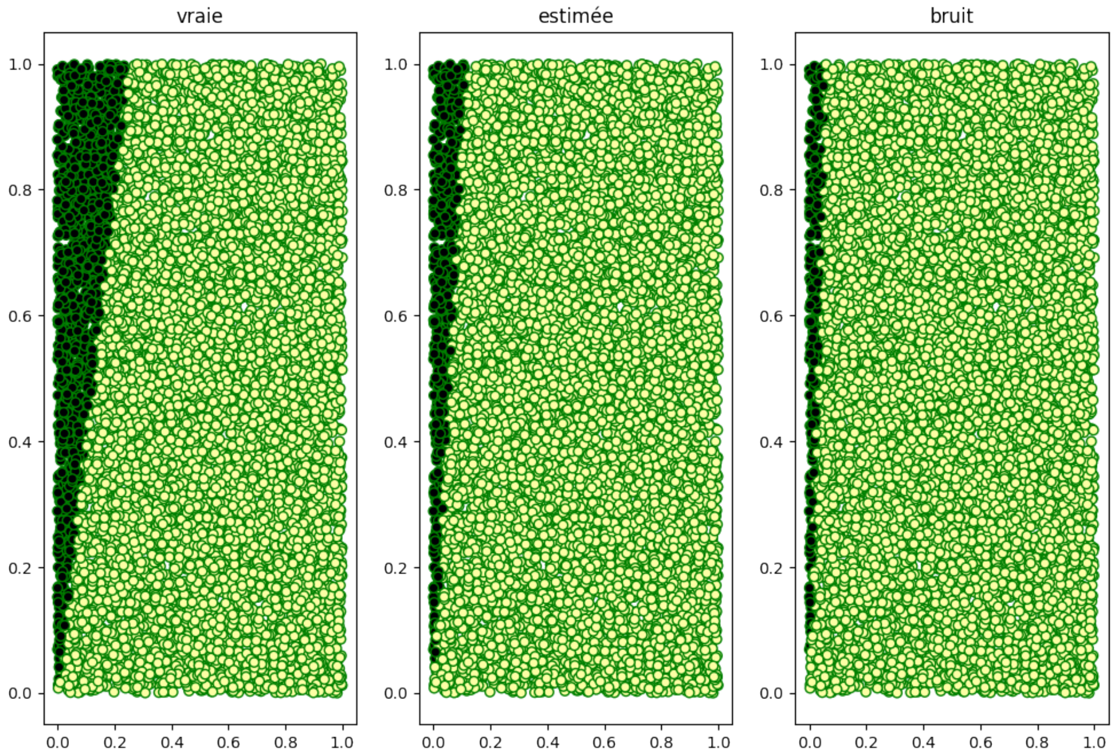
[17]:
```
n=10000
noise=np.random.normal(loc=0,scale=0.125,size=(n,2))
#Comment estimer l'écart-type de la normale? On veut que 95% du bruit soit␣
↪entre -0.25 et 0.25 environ (25% de la variance)
#donc on prend loc=0.125
w_0=np.array([4,-1])
w_0=w_0/np.sqrt(w_0.dot(w_0))
x,y=sample(n,w_0)
x_noised=x+noise
w_est=stochastic_gd(x,y,w_0,alpha,10000)
y_est=x.dot(w_est)
y_est[y_est>0]=1
y_est[y_est<0]=-1
w_noised=stochastic_gd(x_noised,y,w_0,alpha,10000)
y_noised=x.dot(w_noised)
y_noised[y_noised>0]=1
y_noised[y_noised<0]=-1
```

```
print(f"La valeur estimée sans bruit est : {w_est}\n Alors que l'estimation␣
↪avec bruit est (learning rate alpha de {alpha} ): {w_noised}) ")
dist=np.linalg.norm(w_est-w_noised)
print(f"La distance entre les deux vecteurs (bruité versus non bruité) est de␣
↪{dist}")
dist_t=np.linalg.norm(w_noised-w_0)
print(f"La distance entre les deux vecteurs (bruité versus vrai) est de␣
↪{dist_t}")
```

```
La valeur estimée sans bruit est : [ 0.99344162 -0.11434047]
 Alors que l'estimation avec bruit est (learning rate alpha de 0.8 ): [
0.99831147 -0.05808796])
La distance entre les deux vecteurs (bruité versus non bruité) est de
0.05646291037013009
La distance entre les deux vecteurs (bruité versus vrai) est de
0.18658625586090044
```

[18]:
```
ax3=plt.subplot(131)
ax3.scatter(x[:,0],x[:,1],c=y,cmap='inferno',facecolors='none',edgecolors='g')
ax3.set_title("vraie")
ax1=plt.subplot(132)
ax1.scatter(x[:,0],x[:
↪,1],c=y_est,cmap='inferno',facecolors='none',edgecolors='g')
ax1.set_title("estimée")
ax2=plt.subplot(133)
ax2.scatter(x[:,0],x[:
↪,1],c=y_noised,cmap='inferno',facecolors='none',edgecolors='g')
ax2.set_title("bruit")
```

[18]:
```
Text(0.5, 1.0, 'bruit')
```



[19]:
```python
data=pd.read_csv('./data/breast-cancer-wisconsin.data', delimiter=",",␣
↪index_col=False, header=None).replace("?", np.nan).dropna()
print(data.shape)
column_names=["id"]+["x%d" %i for i in range(1,10)]+["class"]
data.columns=column_names
data.index=data.iloc[:,0]
data=data.iloc[:,1:]
data
```

```
(683, 11)
```

[19]:

id	x1	x2	x3	x4	x5	x6	x7	x8	x9	class
1000025	5	1	1	1	2	1	3	1	1	2
1002945	5	4	4	5	7	10	3	2	1	2
1015425	3	1	1	1	2	2	3	1	1	2
1016277	6	8	8	1	3	4	3	7	1	2
1017023	4	1	1	3	2	1	3	1	1	2
…	..	..	..	..	..	..	..	..	..	…
776715	3	1	1	1	3	2	1	1	1	2
841769	2	1	1	1	2	1	1	1	1	2
888820	5	10	10	3	7	3	8	10	2	4
897471	4	8	6	4	3	4	10	6	1	4

```
897471     4   8   8   5   4   5  10   4   1       4

[683 rows x 10 columns]
```

[20]:
```
x=data.iloc[:,0:-1]
y=np.where(data["class"]==2,1,-1)
#y=1 veut dire cancer bénin
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size=0.25,␣
 ↪random_state=0)
scaler = Normalizer().fit(x_train)
normalized_x_train= scaler.transform(x_train)
normalized_x_test = scaler.transform(x_test)
normalized_x_train.shape
```

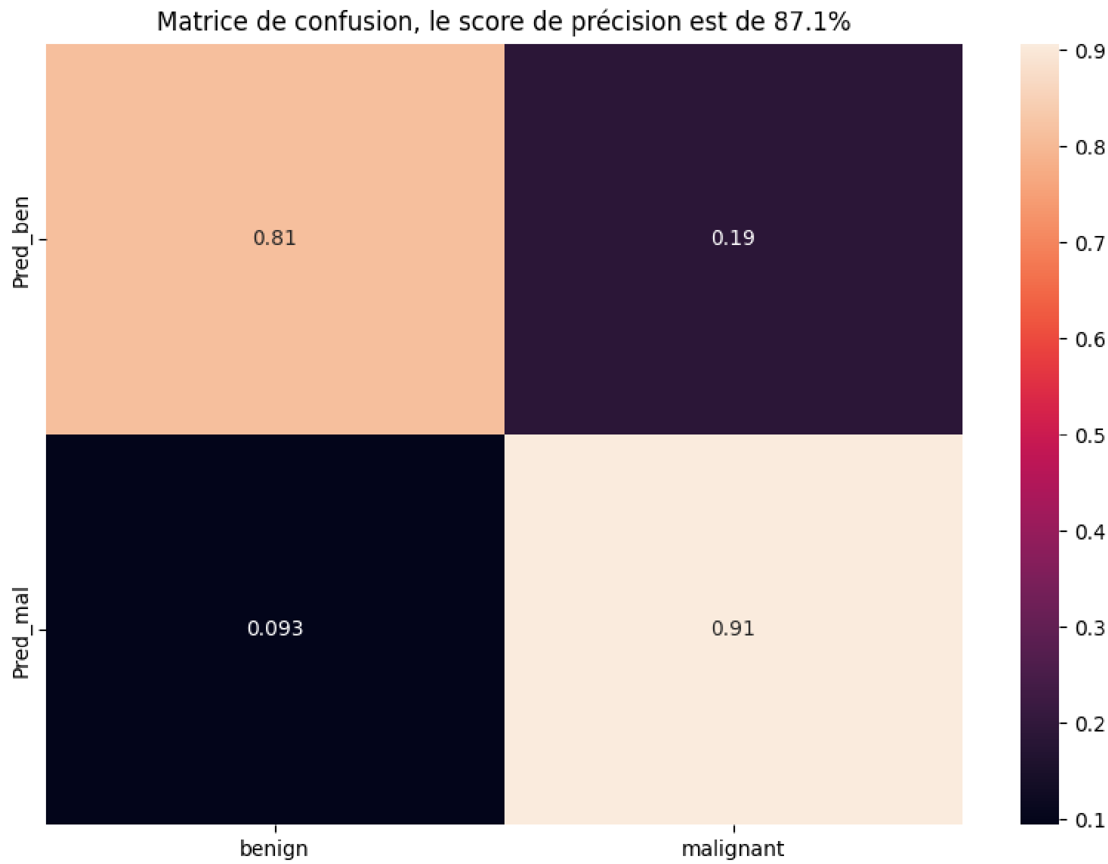
[20]:
```
(512, 9)
```

[21]:
```
w_0=np.random.rand(normalized_x_train.shape[1])
w_0=w_0/(np.sqrt(w_0.dot(w_0)))
w_est=stochastic_gd(normalized_x_train,y_train,w_0,alpha=0.8,niter=100000)



### Confusion matrix
y_pred=np.where(normalized_x_test.dot(w_est)>0,1,-1)
accuracy_score=sklearn.metrics.accuracy_score(y_test,y_pred,normalize=True)
cm=sklearn.metrics.confusion_matrix(y_test, y_pred, normalize='true')
df_cm = pd.DataFrame(cm, columns = ["benign", "malignant"],
                  index = ["Pred_ben", "Pred_mal"])
plt.figure(figsize = (10,7))
sn.heatmap(df_cm, annot=True)
plt.title(f"Matrice de confusion, le score de précision est de␣
 ↪{accuracy_score*100:.1f}%")
```

[21]:
```
Text(0.5, 1.0, 'Matrice de confusion, le score de précision est de 87.1%')
```


Matrice de confusion, le score de précision est de 87.1%
Pred_ben
Pred_mal
0.81
0.19
0.093
0.91
benign
malignant
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1