Quadrinôme :

- ABDESSAMED BOULARIACHE
- KEMOUM Meroua
- MAHIDDINE Mohamed Amine
- TAZIR REDA

TP 4 Regression logistique avec régularisation

Dans ce TP, nous aimerions faire une classification binaire en utilisant la régression.

Pour ce faire, nous étudierons un ensemble de données avec la variable (y) representant la commercialisation d'un profuit et les caractéristiques (X) representant les résultat des tests de qualité test 1 et test 2 du produit.

La prédiction se fera avec l'agorithme de descente du gradient avec régularisation.

Importation des librairies necessaires au travail

Entrée [203]:

```
import numpy as np
import matplotlib.pyplot as plt
```

Lecture des fichiers de données

Pour ce TP, nous allons lire les données à partir d'un fichier csv.

Entrée [204]:

```
# données
data = np.genfromtxt('data.csv', delimiter=',', dtype=float)
data.shape
```

Out[204]:

```
(118, 3)
```

Dans ces données (data), la première colonne represente la première note, la deuxieme colonne la deuxième note et la troisième colonne represente la commercialisation (1 oui 0 non).

Chaque ligne represente un exemple de notre ensemble de données.

Mettons ces données dans leus vecteurs correspondants.

Entrée [205]:

```
# rajoutons l'ordonnée à l'origine theta 0
intercept=np.ones((data.shape[0],1))
X=np.column_stack((intercept,data[:,0:2]))
y = data[:, 2];
# forcer y à avoir une seule colonne
y = y.reshape( y.shape[0], 1)
```

Entrée [206]:

```
print('X', X.shape ,' y ', y.shape)
```

```
X (118, 3)  y  (118, 1)
```

Transformation de données

Dans cette partie, nous aimerions transformer nos données afin d'avoir une fonction polynomiale de degrée 6.

La fonction sera:

$x_0 = 1$

$x_1 = x_1$

$x_2 = x_2$

$x_3 = {x_1}^2$

$x_4 = x_1 x_2$

$x_5 = {x_2}^2$

$x_6 = {x_1}^3$

$x_7 = {x_1}^2 x_2$

$x_8 = x_1 {x_2}^2$

$x_9 = {x_2}^3$

...

Pour un polynme de degrée 6 à 2 variables nous aurons 28 caracteristiques

Question: comment avons nous trouvé ce chiffre?

Astuce: référez vous aux probabilités

$\sum_{i=0}^{p}(i+1) = (\sum_{i=0}^{p} i) + (p+1) = \frac{p(p+1)}{2} + (p+1)$

$\sum_{i=0}^{6}(i+1) = (\frac{6*7}{2}) + 7 = 28$

Entrée [207]:

```
def mapping(X):

    cols = 28
    degree=7
    outX= np.ones((X.shape[0],cols))
    X1=X[:,1]
    X2=X[:,2]
    k=0
    for i in range(degree):
        for j in range(i+1):
            outX[:, k] = np.power(X1,i-j)*(np.power(X2,j));
            k=k+1
    return outX
```

Entrée [208]:

```
X1 = X
X=mapping(X)
X.shape
```

Out[208]:

```
(118, 28)
```

Descente du Gradient : Préparation des fonctions

0- Fonction mpgistique (Sigmoid)

Entrée [209]:

```
def Sigmoid(z):
    # pour une valeur donnée, cette fonction calculera sa sigmoid
    return 1/(1+np.exp(-z));
```

Entrée [210]:

```
k=Sigmoid(-10)
k
```

Out[210]:

```
4.5397868702434395e-05
```

1- Calcul du coût

Cette fonction servira à calculer le cout $J(\theta_0, \theta_1)$

Elle prendra l'ensemble de données d'apprentissage en entrée ainsi que les paramètres définis initialement

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\ \log(h_\theta(x^{(i)})) + (1-y^{(i)})\ \log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Entrée [211]:

```
def computeCostReg(X, y, theta):
    m = len(X)
    theta = np.matrix(theta)
    X = np.matrix(X)
    y = np.matrix(y)

    partie1 = np.multiply(-y, np.log(Sigmoid(X @ theta)))
    partie2 = np.multiply((1 - y), np.log(1 - Sigmoid(X @ theta)))
    reg = (0.0000001 / 2 * m) * np.sum(np.power(theta[:,1:theta.shape[0]], 2))
    J = np.sum(partie1 - partie2) / m + reg

    return J
```

2- Fonction de la descente du gradient

Cette fonction mettra à jour les paramètres θ_0, θ_1 jusqu'à convergence: atteinte du nombre d'itérations max, ou dérivée assez petite.

$$
\begin{aligned}
&\text{Repeat } \{ \\
&\quad \theta_0 := \theta_0 - \alpha \, \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \\
&\quad \theta_j := \theta_j - \alpha \left[\left(\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right] \qquad j \in \{1, 2 \ldots n\} \\
&\}
\end{aligned}
$$

Entrée [212]:

```
def gradientDescent(X, y, theta, alpha, iterations):
    # garder aussi le cout à chaque itération
    # pour afficher le coût en fonction de theta0 et theta1
    m = len(X)
    X = np.matrix(X)
    y = np.matrix(y)
    theta = np.matrix(theta)

    parameters = int(theta.shape[0])
    grad = np.zeros(parameters)
    all_cost = []
    for j in range(iterations):
        error = Sigmoid(X @ theta) - y

        for i in range(parameters):
            term = np.multiply(error, X[:,i])

            if (i == 0):
                grad[i] = np.sum(term) / m
            else:
                grad[i] = (np.sum(term) / m) + ((alpha / m) * theta[i,:])

        g =grad.reshape(theta.shape[0], 1)
        theta[0] = theta[0] - g[0]
        theta[1:,0] = theta[1:,0] - g[1:]
        all_cost.append(computeCostReg(X, y, theta)) # garder aussi le cout à chaque itérat

    return grad, all_cost
```

Descente du Gradient : Appel des fonctions

Initialisation de θ_0 et θ_1

Entrée [213]:

```
n=X.shape[1]
theta = np.zeros((n, 1))
theta
```

Out[213]:

```
array([[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.]])
```

Calculer le cout initial

Entrée [214]:

```
initialCost=computeCostReg(X, y, theta)
print(initialCost)
```

```
0.6931471805599454
```

Appel des la fonction de calcul du gradient

Entrée [215]:

```
# paramètres
iterations = 1500;
alpha = 0.01;

# paramètre de regression
lambdaa = 1;

# Appel
theta, allcost = gradientDescent(X, y, theta, alpha, iterations);
```

Entrée [226]:

```
theta = theta.reshape(-1, 1)
print(theta)
```

```
[[-3.36060117e-04]
 [-1.77290605e-04]
 [-4.27657363e-04]
 [ 1.95898121e-04]
 [ 8.23174723e-04]
 [ 1.07123935e-03]
 [-2.12071635e-04]
 [ 4.48686581e-05]
 [-5.36827709e-04]
 [ 3.80006791e-04]
 [ 3.98160343e-04]
 [-6.91255993e-04]
 [ 4.16777415e-04]
 [ 4.20143520e-04]
 [ 3.65641427e-04]
 [ 2.70043913e-04]
 [-6.67451801e-05]
 [-7.11290059e-04]
 [ 5.06768525e-04]
 [ 4.11833497e-04]
 [-5.82437423e-04]
 [ 6.12864841e-04]
 [-3.66245247e-04]
 [ 1.04856477e-04]
 [-3.81787007e-04]
 [ 4.44497049e-04]
 [ 5.41892582e-04]
 [-5.87075124e-04]]
```

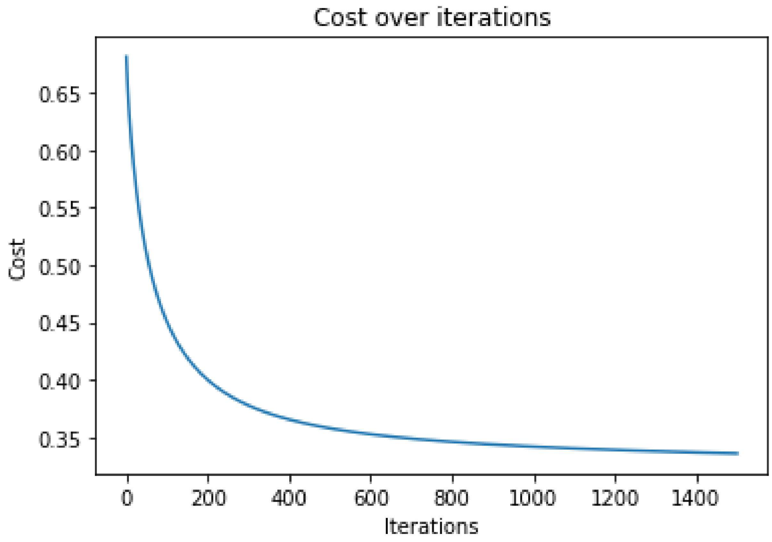
Traçage de la fonction du coût

Entrée [217]:

```
plt.plot(allcost)

#labels
plt.title('Cost over iterations')
plt.xlabel("Iterations")
plt.ylabel("Cost")
```

Out[217]:

```
Text(0, 0.5, 'Cost')
```



Notons que $\theta^T x$ est équivalent à $X\theta$ où $X = \begin{pmatrix} ..(x^{(1)})^T.. \\ ..(x^{(2)})^T.. \\ . \\ . \\ . \\ ..(x^{(m)})^T.. \end{pmatrix}$

Dessin de la limite de decision (Descision Boundary)

Dans cette partie, nous aimerions dessiner la ligne separatrice d nos données

Entrée [218]:

```
def drawCircle(X, y, theta):
    x1, x2 = np.meshgrid(np.linspace(X[:, 1].min(), X[:, 1].max(), 100),np.linspace(X[:, 2]
    i = (mapping(np.c_[np.ones(x1.size), x1.ravel(), x2.ravel()]) @ theta).reshape(x1.shape

    print(np.amin(i), np.amax(i))

    plt.contour(x1, x2, i, levels=[0], colors= "yellow", linestyles="dashed")
```

Classification (Prédiction)

Ici il serait interessant de calculer la prédiction en utilisant un seuil i.e. si h>seuil alors classe =1 sinon classe = 0

Entrée [219]:

```
def predict(X, theta):
    probability = Sigmoid(X @ theta)
    y_pred = [1 if x >= 0.5 else 0 for x in probability]
    return y_pred
```

Affichage

Graphe representant les acceptations selon les caracteristiques

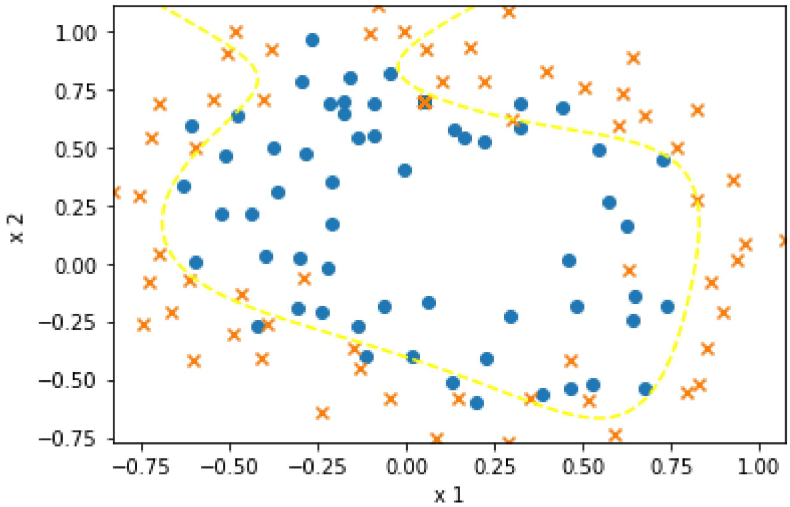
Entrée [220]:

```
plt.scatter(X[np.where(y==1),1],X[np.where(y==1),2], label="y=1",marker ='o')
plt.scatter(X[np.where(y==0),1],X[np.where(y==0),2], label="y=0",marker ='x')
drawCircle(X, y, theta)
plt.xlabel('x 1')
plt.ylabel('x 2')
```

```
-0.0008863340391108937 0.002689975957274041
```

Out[220]:

```
Text(0, 0.5, 'x 2')
```



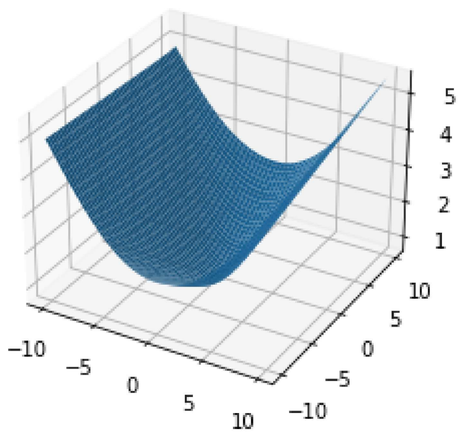
Traçage du coût en fonction de theta0 et theta1

Entrée [221]:

```
theta_0_range, theta_1_range = np.meshgrid(np.arange(theta[0] - 10, theta[0] + 10, 0.1),
                                           np.arange(theta[1] - 10, theta[1] + 10, 0.1))
theta_mesh = np.expand_dims(np.stack((theta_0_range, theta_1_range), axis=-1), axis=3)

liste0 =list()

for i in range(theta_mesh.shape[0]):
    l=list()
    for j in range(theta_mesh.shape[1]):
        l.append(computeCostReg(X, y, np.concatenate((theta_mesh[i, j], theta[2:]), axis=0)
    liste0.append(l)

liste = np.asarray(liste0)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(theta_0_range, theta_1_range, liste)
```

Out[221]:

```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x19000b89150>
```



Qualité du classifieur

Prédire des valeurs de y

Ici il serait interessant de calculer la précision de notre classifieur

Essayons de calculer ça avec

moyenne(y==y-pred) * 100

Ceci donnera un pourcentage de precision

Entrée [222]:

```
# calcul de precision = nombre de valeurs bien prédites (ici sur toute la base X)
y_pred=predict(X, theta)
precision = np.mean(y==y_pred)*100
precision
```

Out[222]:

```
50.08618213157139
```

Vérification de l'implementation

Comparer vos algorithmes à ceux de scikitlearn

Entrée [223]:

```
from sklearn.linear_model import LogisticRegression

X2 = X[:,1:]
y2 = np.squeeze(y, axis=1)

# use LogisticRegression
log_reg = LogisticRegression(penalty='l2')
log_reg.fit(X2, y2)

theta_sklearn = np.zeros(theta.shape)
# Coefficient of the features in the decision function. (from theta 1 to theta n)
theta_sklearn[1:] = log_reg.coef_.reshape(-1, 1)#.ravel()

# Intercept (a.k.a. bias)  (theta 0)
theta_sklearn[0] = log_reg.intercept_[0]

theta_sklearn = np.reshape(theta_sklearn, (-1, 1))
```

Entrée [224]:

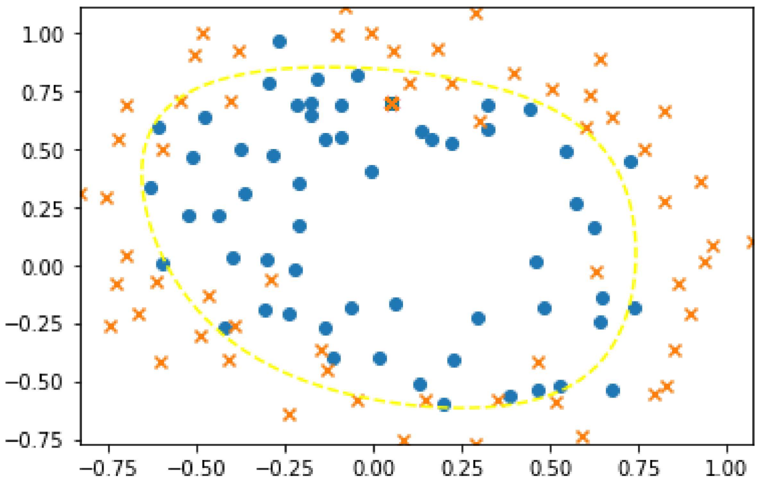
```
print("thetas :")
print(theta_sklearn[1:])
print("Intercept :")
print(theta_sklearn[0])
```

```
thetas :
[[ 0.62536719]
 [ 1.18095854]
 [-2.01961804]
 [-0.91752388]
 [-1.43170395]
 [ 0.12391867]
 [-0.36536954]
 [-0.35715555]
 [-0.17501434]
 [-1.45827831]
 [-0.05112356]
 [-0.61575808]
 [-0.27472128]
 [-1.19276292]
 [-0.24241519]
 [-0.20587922]
 [-0.0448395 ]
 [-0.27780311]
 [-0.29535733]
 [-0.45625452]
 [-1.04347339]
 [ 0.02770608]
 [-0.29252353]
 [ 0.01550105]
 [-0.32746466]
 [-0.1439423 ]
 [-0.92460358]]
Intercept :
[1.27271075]
```

Entrée [225]:

```
plt.figure()

plt.scatter(X[np.where(y==1),1],X[np.where(y==1),2], label="y=1",marker ='o')
plt.scatter(X[np.where(y==0),1],X[np.where(y==0),2], label="y=0",marker ='x')

drawCircle(X1, y, theta_sklearn)
```

```
-14.906585165164152 1.4926201404230859
```



Renforcement d'apprentissage

Mettre ici toute idée qui pourrait renforcer votre apprentissage

Il existe différentes approches pour optimiser la fonction de coût :

- Conjugate gradient
- BFGS
- L-BFGS

Ce sont des exemples d'algorithmes d'optimisation plus sophistiqués que la descente de gradient qui permettent d'évoluer beaucoup mieux les plus gros problèmes d'apprentissage.

Consignes

Le travail est à remettre par groupe de 4 au maximum [1..4].

Le délai est le vendredi 01 Avril 2022 à 22h

Entrée [61]:

```
# bonne chance
```