Quadrinôme :

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

TP 2 Descente du Gradient

Dans ce TP, nous aimerions prédire le bénéfice d'une entreprise dans plusieurs ville en nous basant sur les habitant de cette ville.

Pour ce faire, nous étudierons un ensemble de données avec le bénéfice (y) et les caractéristiques des habitants (X).

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

Importation des librairies necessaires au travail

Entrée [1]:

```
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 [2]:

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

Out[2]:

```
(97, 2)
```

Dans ces données (data), la première colonne represente la première caractéristique (la population d'une ville x1000), la deuxieme colonne represente le bénéfice (x1000).

Chaque ligne represente un exemple de notre ensemble de données.

Mettons ces données dans leus vecteurs correspondants.

Entrée [3]:

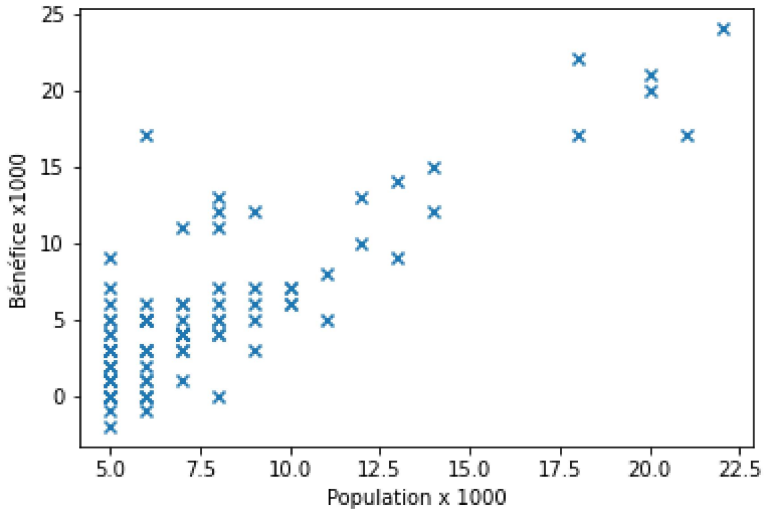
```
# rajoutons l'ordonnée à l'origine theta 0
intercept=np.ones((data.shape[0],1))
X=np.column_stack((intercept,data[:,0]))
y = data[:, 1];
```

Entrée [4]:

```
# traçons ces données pour visualisation
plt.scatter(X[:,1],y,marker ='x')
plt.xlabel('Population x 1000')
plt.ylabel('Bénéfice x1000')
```

Out[4]:

```
Text(0, 0.5, 'Bénéfice x1000')
```



Descente du Gradient : Préparation des fonctions

Cost function

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

$$J(\theta) = \frac{1}{2m} (X\theta - y)^T (X\theta - y) \text{ (vectorized version)}$$

Gradient

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} X^T (X\theta - y)$$

- 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

Entrée [5]:

```
def computeCostNonVect(X, y, theta):
    m = len(y)
    sum = 0

    for i in range(0, m-1):
        h=(X[i][0])*theta[0][0]+X[i][1]*theta[1][0]
        s=(h-y[i])**2
        sum=sum+s

    return (1/(2*m))*sum
    # idéalement, tracer le coût à chaque itération pour s'assurer que la descente du gra
    # calculer le coût SANS vectorisation,
    # comparer le temps de traitement
```

Entrée [6]:

```
def computeCost(X, y, theta):
    # calculer le coût AVEC vectorisation,
    # comparer le temps de traitement

    m = y.shape[0] #the number of training examples
    return np.sum(np.power((np.dot(X, theta) - y), 2)) / (2 * m)
```

- 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.

Gradient descent algorithm

repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot x^{(i)}$$

}

Entrée [7]:

```
def gradientDescent(X, y, theta, alpha, iterations):

    m = y.shape[0] #len(X)
    all_cost = []
    all_theta = [theta]
    for i in range(iterations):
        all_cost.append(computeCost(X, y, all_theta[-1])) # garder aussi le cout à chaque i
        all_theta.append(all_theta[-1] + float(alpha) * (np.dot(X.T, (y - np.dot(X,all_thet
    return all_theta,all_cost
```

Descente du Gradient : Appel des fonctions

Initialisation de θ_0 et θ_1

Entrée [8]:

```
theta = np.zeros((2, 1))
```

Calculer le cout initial

Entrée [9]:

```
y = y.reshape(-1, 1) #transposer y pour être de la forme (97, 1), afin d'avoir le mm nbr de
import time
start = time.time()

initialCost=computeCost(X, y, theta)

end = time.time()
elapsed = end - start

print(f'Temps d\'exécution : {elapsed:.2}ms')

##---------------
start = time.time()

initialCost2=computeCostNonVect(X, y, theta)

end = time.time()
elapsed = end - start
print(f'Temps d\'exécution : {elapsed:.2}ms')

print(initialCost)
print(initialCost2)
```

```
Temps d'exécution : 0.001ms
Temps d'exécution : 0.002ms
29.257731958762886
[29.25773196]
```

Appel des la fonction de calcul du gradient

on remarque que la fonction computeCost est plus rapide que la fonction computeCostNonVect

Entrée [10]:

```
# paramètres
iterations = 1500;
alpha = 0.01;
# Appel
theta, cost = gradientDescent(X, y, theta, alpha, iterations);
```

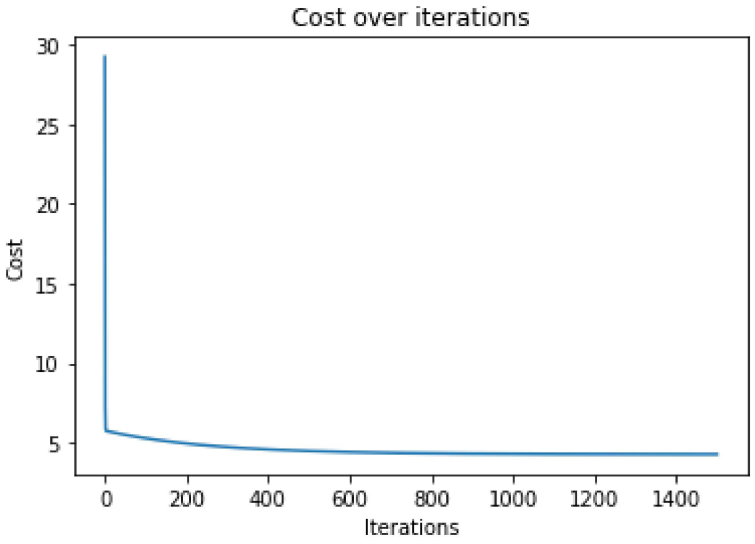
Traçage de la fonction du coût

Entrée [11]:

```
plt.plot(cost)

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

Out[11]:

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



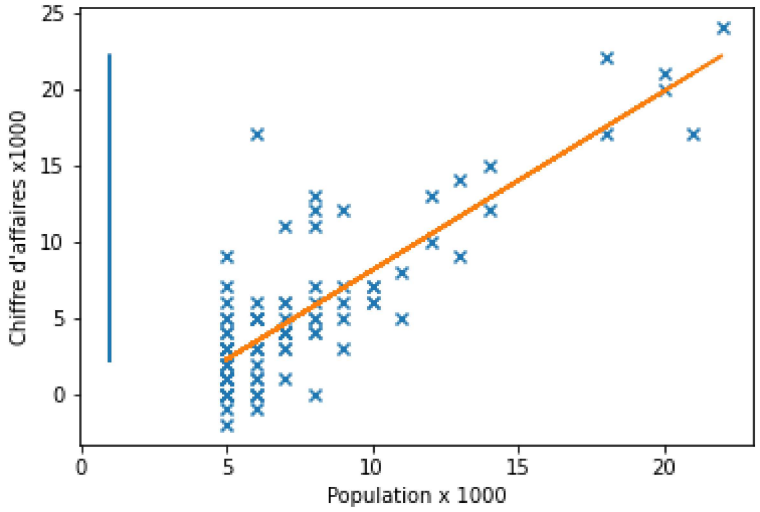
Notons que $\theta^T x$ est équivalent à $X\theta$ où $X = \begin{pmatrix} .. (x^{(1)})^T .. \\ .. (x^{(2)})^T .. \\ . \\ . \\ . \\ .. (x^{(m)})^T .. \end{pmatrix}$

Entrée [12]:

```
best_theta = theta[-1]
y_pred=np.matmul(X,best_theta)
plt.scatter(X[:,1],y,marker ='x')
plt.xlabel('Population x 1000')
plt.ylabel('Chiffre d\'affaires x1000')
plt.plot(X,y_pred)
# la ligne du graphe represente le traçage de la fonction hypothèse
# la ligne devrait se rapprocher des données après entrainement avec la descente du gredien
```

Out[12]:

```
[<matplotlib.lines.Line2D at 0x205a14ca170>,
 <matplotlib.lines.Line2D at 0x205a14ca110>]
```

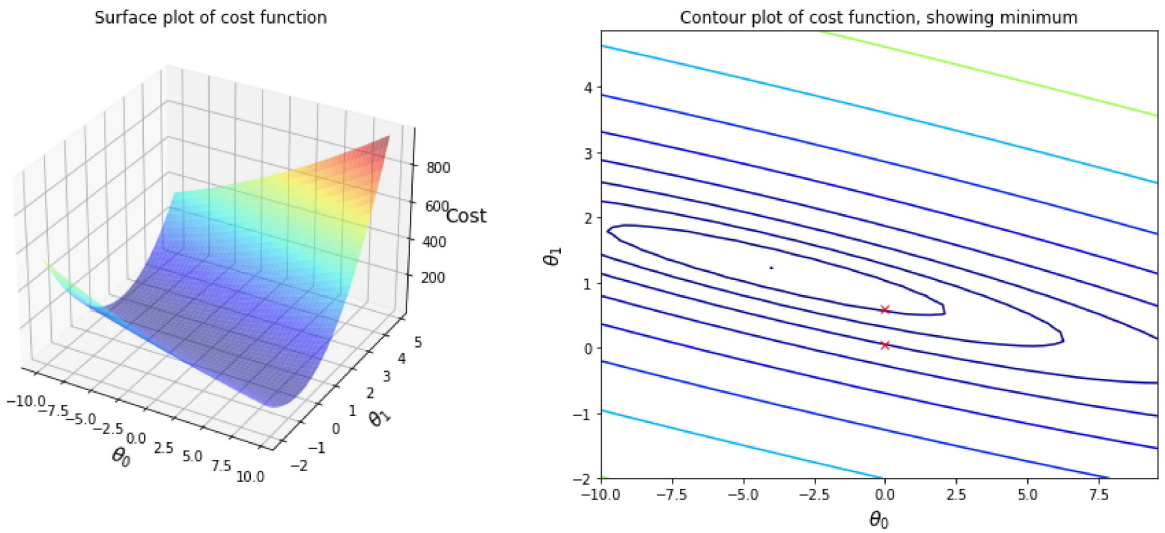


Traçage du coût en fonction de theta0 et theta1

Entrée [13]:

```
print('Visualizing J(theta_0, theta_1) ...')

from mpl_toolkits.mplot3d import axes3d

# Create meshgrid.
xs = np.arange(-10, 10, 0.4)
ys = np.arange(-2, 5, 0.14)
xx, yy = np.meshgrid(xs, ys)

# Initialize J values to a matrix of 0's.
J_vals = np.zeros((xs.size, ys.size))

# Fill out J values.
for index, v in np.ndenumerate(J_vals):
    J_vals[index] = computeCost(X, y, [[xx[index]], [yy[index]]])

# Create a set of subplots.
fig = plt.figure(figsize=(16, 6))
ax1 = fig.add_subplot(121, projection='3d')
ax2 = fig.add_subplot(122)

# Create surface plot.
ax1.plot_surface(xx, yy, J_vals, alpha=0.5, cmap='jet')
ax1.set_zlabel('Cost', fontsize=14)
ax1.set_title('Surface plot of cost function')

# Create contour plot.
ax2.contour(xx, yy, J_vals, np.logspace(-2, 3, 20), cmap='jet')
ax2.plot(theta[0], theta[1], 'rx')
ax2.set_title('Contour plot of cost function, showing minimum')

# Create labels for both plots.
for ax in fig.axes:
    ax.set_xlabel(r'$\theta_0$', fontsize=14)
    ax.set_ylabel(r'$\theta_1$', fontsize=14)
```

```
Visualizing J(theta_0, theta_1) ...
```



Prédire des valeurs de y

Entrée [14]:

```
# Predire pour une population = 35,000 et 70,000
predict1 = np.matmul([1, 3.5],best_theta)

print("pour une Population de 35,000 , On prédit un Profit de ",predict1*1000)
```

```
pour une Population de 35,000 , On prédit un Profit de  [498.98053183]
```

Entrée [15]:

```
predict2 = np.matmul([1, 7],best_theta)

print("pour une Population de 70,000 , On prédit un Profit de ",predict2*1000)
```

```
pour une Population de 70,000 , On prédit un Profit de  [4597.6771742]
```

Régression linéaire à plusieurs variables

Proposer, l'aide des fonctions définies precedemment, une regression linéaire lorsque le nombre de descripeurs est supérieur à 1, en utilisant la base d'apprentissage suivante

Entrée [16]:

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

Out[16]:

```
(47, 3)
```

Pour cette ensemble de données, nous voudrions prédire le prix d'une maison (3ème colonne de dataMulti) à partir de :

- sa supérficie (1ère colonne)
- son nombre de chambres (2ème colonne)

Entrée [17]:

```
# d'abord créer X et y
intercept=np.ones((dataMulti.shape[0],1))
cols = dataMulti.shape[1] # recupérer le nombre de colonnes
X1 = np.column_stack((intercept,dataMulti[:,0:cols-1]))
y1 = dataMulti[:, cols-1:cols]
```

Entrée [18]:

```
# Pas besoins de redifinir nos fonctions de coût, le cas de plusieurs variables est prit en

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

Mise à l'échelle Et Normalisation des données en moyenne = 0 :

The data can be normalized by subtracting the mean (μ) of each feature and a division by the standard deviation (σ). This way, each feature has a mean of 0 and a standard deviation of 1. This results in faster convergence.

$$x := \frac{x - \mu}{\sigma}$$

Entrée [19]:

```
data_mean = dataMulti.mean()
data_std = dataMulti.std()
normalized_data = (dataMulti - data_mean) / data_std
```

Entrée [20]:

```
intercept=np.ones((normalized_data.shape[0],1))
cols = normalized_data.shape[1] # recupérer le nombre de colonnes
X2 = np.column_stack((intercept,normalized_data[:,0:cols-1]))
y2 = normalized_data[:, cols-1:cols]
```

Appliquer la descente du gradient à plusieurs variables

AVEC Normalisation :

Entrée [21]:

```
# initialiser theta
theta2 = np.zeros((cols, 1))

# Appliquer la descente du gradient
theta2, cost2 = gradientDescent(X2, y2, theta2, alpha, iterations)
best_theta2 = theta2[-1]

# Calculer le cout initial
computeCost(X2, y2, theta2[-1])
```

Out[21]:

```
0.2503474286902239
```

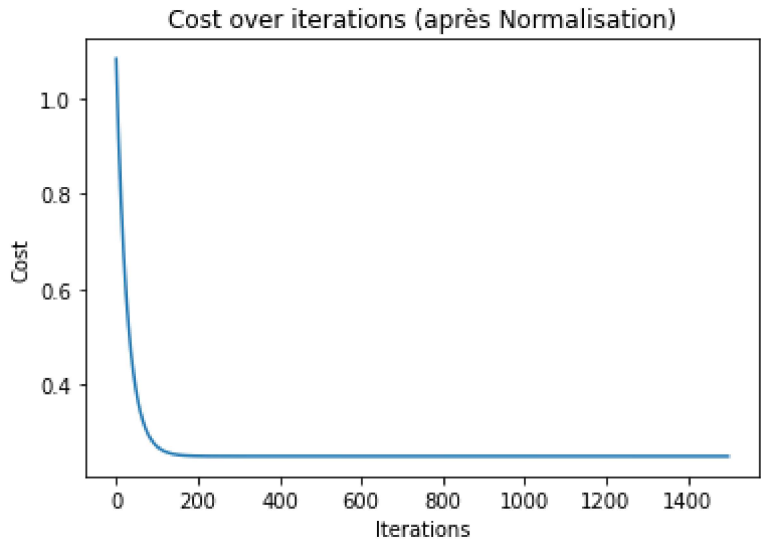
- Traçage de la fonction du coût

Entrée [22]:

```
plt.plot(cost2)

#Labels
plt.title('Cost over iterations (après Normalisation)')
plt.xlabel("Iterations")
plt.ylabel("Cost")
```

Out[22]:

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



Entrée [23]:

```
start = time.time()

initialCost=computeCost(X2, y2, theta2[-1])

end = time.time()
elapsed = end - start

print(f'Temps d\'exécution : {elapsed:.2}ms')
```

```
Temps d'exécution : 0.0ms
```

Comparaison de la descente du gradient avec et sans normalisation

- **Exemple 1**: voir la difference dans le temps de calcul

le Temps d'exécution pour sans Normalisation est 0.002ms qu'est plus grand que le Temps d'exécution : 0.0ms avec Normalisation

Vérification de l'implementation

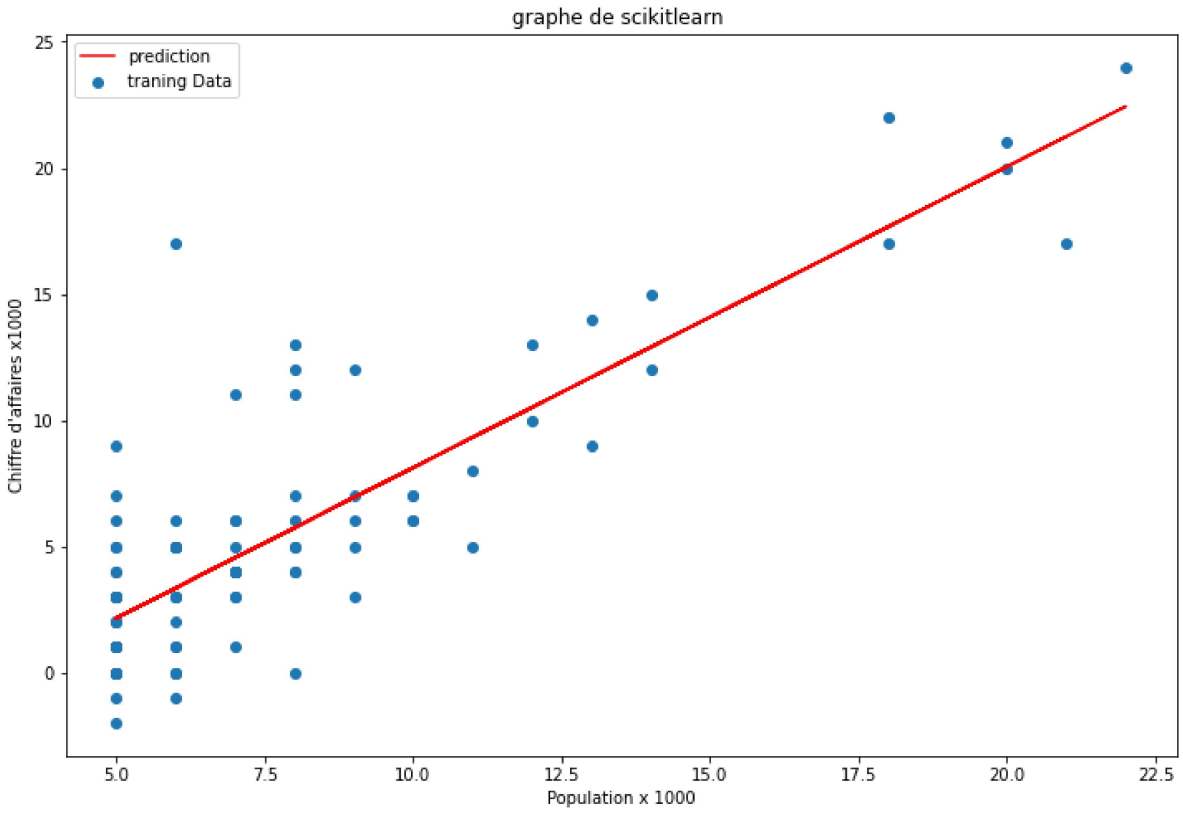
Comparer vos algorithmes à ceux de scikitlearn

Entrée [24]:

```
from sklearn import linear_model
model = linear_model.LinearRegression()
model.fit(X, y)

x = np.array(X[:, 1].T)

f = model.predict(X).flatten()

fig, ax = plt.subplots(figsize=(12,8))
ax.plot(x, f, 'r', label='prediction')
ax.scatter(X[:,1], y, label='traning Data')
#Labels
ax.set_xlabel('Population x 1000')
ax.set_ylabel('Chiffre d\'affaires x1000')
ax.set_title('graphe de scikitlearn ')
ax.legend(loc=2)
```

Out[24]:

```
<matplotlib.legend.Legend at 0x205b1eb2080>
```



- On voit bien que le graphe de la 1ere partie et celui de la lib sont similaire

Renforcement d'apprentissage

Mettre ici toute idée qui pourrait renforcer votre apprentissage

Nous pouvons améliorer nos features et la forme de notre fonction d'hypothèse de différentes manières,

- En combinant les différents features en un seul. par exemple, on peut combiner x_1 et x_2 en une nouvelle feature x_3 (tq : x_3 = x_1 * x_2)

- Nous pouvons modifier le comportement de la courbe de notre fonction d'hypothèse en la transformant en une fonction quadratique, cubique ou racine carrée (ou autre).

Consignes

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

Le délai est le vendredi 18 Mars 2022 à 22h

Entrée [25]:

```
# bonne chance
```