Université de Thiès
UFR SES
Master en Science des Données et Applications / Option IA
Deep Learning (CM : Pr M.BOUSSO – M.LANDU)

# Rapport Projet Régression Linéaire Multiple

## # Exercice 1: Régression Linéaire

### Importation des librairies a utiliser

```
import import_ipynb

# used for manipulating directory paths

import os


# Scientific and vector computation for python

import numpy as np


# Plotting library

from matplotlib import pyplot as plt

from mpl_toolkits.mplot3d import Axes3D  # needed to plot 3-D surfaces


# library written for this exercise providing additional functions for assignment submission, and others


import utils


# tells matplotlib to embed plots within the notebook

%matplotlib inline
```

# Création d'une Simple python and `numpy` function

```
def warmUpExercise(n):
    A = np.eye(n)
 return A
```

**Appel de la fonction**

```
A=warmUpExercise(5)
print("Matrice d'identité d'ordre 5 ")
print(A)
```

**Résultat :**

```
Matrice d'identité d'ordre 5
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

---

# 2 Régression linéaire à une seule variable

**Code**

```
# Read comma separated data
data = np.loadtxt(os.path.join('Data', 'ex1data1.txt'),delimiter=',')
X = data[:,0]
y = data[:,1]
m = X.size  # number of training examples
# print out some data points
print('{:>8s}{:>10s}'.format('Pop(X)', 'profit(y)'))
print('-'*26)
for i in range(10):
```

print('{:8.0f}{:10.0f}'.format(X[i], y[i]))

***Résultat***

```
  Pop(X) profit(y)
------------------------
       6         18
       6          9
       9         14
       7         12
       6          7
       8         12
       7          4
       9         12
       6          7
       5          4
```

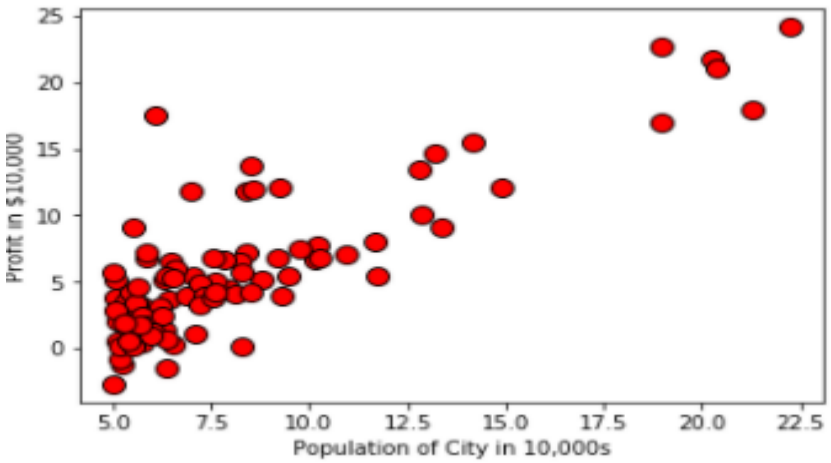
## 2.1 Creation fonction Plotting the Data

```
def plotData(x, y,xlabel,ylabel)
    from matplotlib import pyplot as plt
    fig = plt.figure()  # open a new figure
    plt.plot(x, y, 'ro', ms=10, mec='k')
    plt.ylabel('Profit in $10,000')
    plt.xlabel('Population of City in 10,000s')
```

**Appel de la fonction**

```
plotData(X, y,'Population of City in 10,000s','Profit in $10,000')
```

**Resulat :**




---

## 2.2.3 Calcul de la fonction coût ($\theta$)

```
def computeCost(X, y, theta):
# initialize some useful values
    m=len(y)
    # You need to return the following variables correctly
    J =1/(2*m)*np.sum((model(X,theta)-y)**2)
     return J
```

**Appel de fonction**

```
J = computeCost(X, y, theta=np.array([0.0, 0.0]))
print('With theta = [0, 0] \nCost computed = %.2f' % J)
print('Expected cost value (approximately) 32.07\n')
# further testing of the cost function
J = computeCost(X, y, theta=np.array([-1, 2]))
print('With theta = [-1, 2]\nCost computed = %.2f' % J)
print('Expected cost value (approximately) 54.24')
```

**Résultat**

```
With theta = [0, 0]
Cost computed = 32.07
Expected cost value (approximately) 32.07

With theta = [-1, 2]
Cost computed = 54.24
Expected cost value (approximately) 54.24
```

## *2.2.4 Gradient descent*

```
def grad(X, y, theta):
```

```
    m = len(y)
    return 1/m * X.T.dot(model(X, theta) - y)
def gradientDescent(X, y, theta, alpha, num_iters):
    theta = theta.copy()
    J_history = [] # Use a python list to save cost in every iteration
    J_history=np.zeros(num_iters)
    for i in range (0,num_iters):
        theta=theta-alpha*grad(X,y,theta)
        J_history[i]=computeCost(X,y,theta)
    # save the cost J in every iteration
    return theta, J_history
```

**Appel de fonction**

```
# initialize fitting parameters
theta = np.zeros(2)

# some gradient descent settings
iterations = 1500
alpha = 0.01

theta, J_history = gradientDescent(X ,y, theta, alpha, iterations)
print('Theta found by gradient descent: {:.4f}, {:.4f}'.format(*theta))
print('Expected theta values (approximately): [-3.6303, 1.1664]')
```

**Résultat**

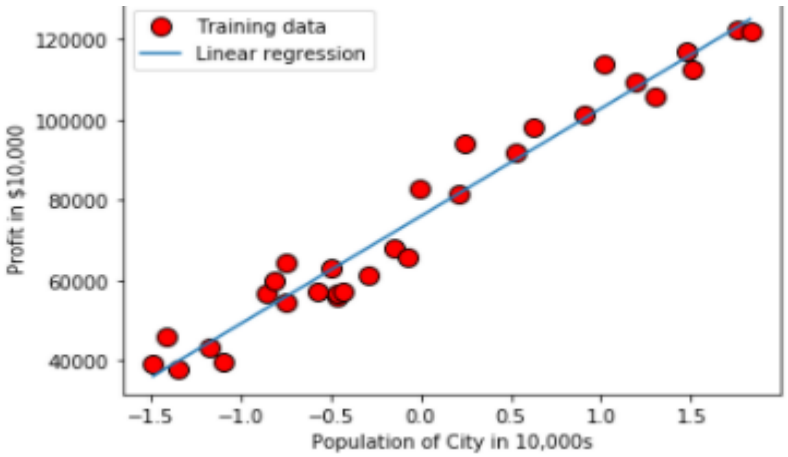
```
Theta found by gradient descent: -3.6303, 1.1664
Expected theta values (approximately): [-3.6303, 1.1664]
```

## # plot the linear fit

plotData(X[:, 1], y,'Population of City in 10,000s',"Profit in $10000")

plt.plot(X[:, 1], np.dot(X, theta), '-')

plt.legend(['Training data', 'Linear regression']);




---

**# Predict values for population sizes of 35,000 and 70,000**

predict1 = np.dot([1, 3.5], theta)

print('For population = 35,000, we predict a profit of {:.2f}\n'.format(predict1*10000))

predict2 = np.dot([1, 7], theta)

print('For population = 70,000, we predict a profit of {:.2f}\n'.format(predict2*10000))

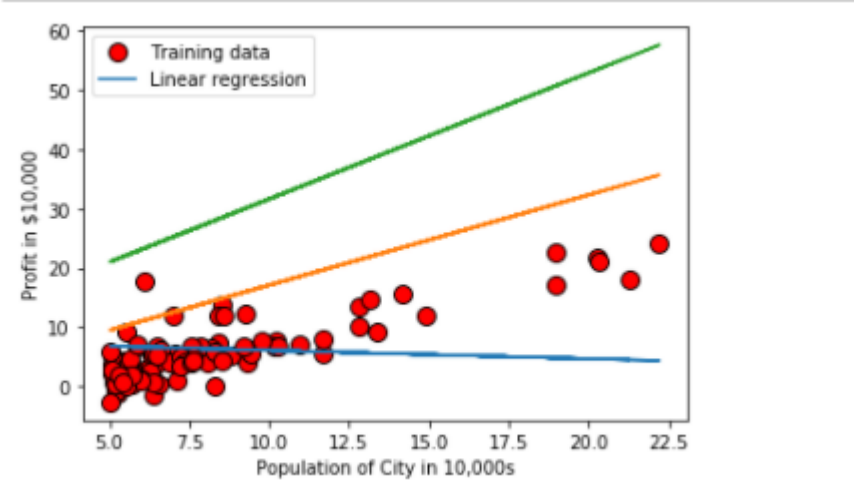
```
For population = 35,000, we predict a profit of 1698657108.94

For population = 70,000, we predict a profit of 2637284433.48
```

**Exercice: Choisir 3 points dans le contour et représenter les droites qui leur correspondent dans le nuage de points**

**Code**

from matplotlib import pyplot

```
# plot the linear fit

plotData(X[:, 1], y,'Population of City in 10,000s',"Profit in $10000")

for i in range(3):

    theta=[np.random.choice(theta0_vals), np.random.choice(theta1_vals)]

    pyplot.plot(X[:, 1], np.dot(X, theta), '-')

plt.legend(['Training data', 'Linear regression']);
```




## Exercice optionnel:

On Reprend les mêmes cellules en changeant de cost function, prenez la moyenne des valeurs absolues des erreurs ( MAE).

**Code**

```
def model(X, theta):

    return X.dot(theta)

def computeCostAbsolute(X, y, theta):

    J =(1/m)*np.sum(np.abs(model(X,theta)-y))

    return  J
```

**Appel  Fonction**

```
J = computeCostAbsolute(X, y, theta=np.array([0.0, 0.0]))

print('With theta = [0, 0] \nCost computed = %.2f' % J)
```

print('Expected cost value (approximately) 76003.00\n')

# further testing of the cost function

J = computeCostAbsolute(X, y, theta=np.array([-1, 2]))

print('\n\nWith theta = [-1, 2]\nCost computed = %.2f' % J)

print('Expected cost value (approximately) 76004.00\n')

## Résultat

```
Entrée [145]: J = computeCostAbsolute(X, y, theta=np.array([0.0, 0.0]))
              print('With theta = [0, 0] \nCost computed = %.2f' % J)
              print('Expected cost value (approximately) 76003.00\n')
              # further testing of the cost function
              J = computeCostAbsolute(X, y, theta=np.array([-1, 2]))
              print('\n\nWith theta = [-1, 2]\nCost computed = %.2f' % J)
              print('Expected cost value (approximately) 76004.00\n')

              With theta = [0, 0]
              Cost computed = 76003.00
              Expected cost value (approximately) 76003.00



              With theta = [-1, 2]
              Cost computed = 76004.00
              Expected cost value (approximately) 76004.00
```

# Régression linéaire avec plusieurs variables

Dans cette partie, nous avons implémenter une régression linéaire avec plusieurs variables pour prédire les prix des maisons.

## Feature Normalization

# Load data

data = np.loadtxt(os.path.join('Data', 'ex1data2.txt'), delimiter=',')

X = data[:, :2]

y = data[:, 2]

m = y.size

# print out some data points

print('{:>8s}{:>8s}{:>10s}'.format('X[:,0]', 'X[:, 1]', 'y'))

print('-'*26)

for i in range(10):

  print('{:8.0f}{:8.0f}{:10.0f}'.format(X[i, 0], X[i, 1], y[i]))

```
# Load data
data = np.loadtxt(os.path.join('Data', 'ex1data2.txt'), delimiter=',')
X = data[:, :2]
y = data[:, 2]
m = y.size

# print out some data points
print('{:>8s}{:>8s}{:>10s}'.format('X[:,0]', 'X[:, 1]', 'y'))
print('-'*26)
for i in range(10):
    print('{:8.0f}{:8.0f}{:10.0f}'.format(X[i, 0], X[i, 1], y[i]))
```

```
  X[:,0] X[:, 1]        y
--------------------------
    2104       3   399900
    1600       3   329900
    2400       3   369000
    1416       2   232000
    3000       4   539900
    1985       4   299900
    1534       3   314900
    1427       3   198999
    1380       3   212000
    1494       3   242500
```

---

On compléte le code dans la fonction `featureNormalize`:

**def  featureNormalize(X):**

```
  # You need to set these values correctly
  X_norm = X.copy()
  mu = np.zeros(X.shape[1])
```

```
    sigma = np.zeros(X.shape[1])
    n = X_norm.shape[1]

    for i in range(n):
        mu[i]=np.mean(X[:,i])
        sigma[i]=np.std(X[:,i])
        for j in range(len(X_norm)):
            X_norm[j,i]=(X_norm[j,i]-mu[i])/sigma[i]

    return X_norm, mu, sigma
```

**# call featureNormalize on the loaded data**

```
X_norm, mu, sigma = featureNormalize(X)

print('Computed mean:', mu)
print('Computed standard deviation:', sigma)
```

**# Computed mean: [2000.68085106    3.17021277]**
**# Computed standard deviation: [7.86202619e+02 7.52842809e-01**

## Resultat :

```
Computed mean: [2000.68085106    3.17021277]
Computed standard deviation: [7.86202619e+02 7.52842809e-01]
```

# Gradient Descent

On compléte le code des fonctions computeCostMulti et gradientDescentMulti pour implémenter la fonction de coût et la descente de gradient pour la régression linéaire avec plusieurs variables

```
def model(X, theta):

    return X.dot(theta)

def computeCostMulti(X, y, theta):

    """

    Compute cost for linear regression with multiple variables.

    Computes the cost of using theta as the parameter for linear regression to fit the data points in X and y.
```

```
    Parameters
    ----------
    X : array_like
        The dataset of shape (m x n+1).

    y : array_like
        A vector of shape (m, ) for the values at a given data point.

    theta : array_like
        The linear regression parameters. A vector of shape (n+1, )

    Returns
    -------
    J : float
        The value of the cost function.

    Instructions
    ------------
    Compute the cost of a particular choice of theta. You should set J to the cost.
    """
    # Initialize some useful values
    m = y.shape[0] # number of training examples
    # You need to return the following variable correctly
    J = 0
    J = (0.5/m) * np.sum((model(X, theta) - y)**2)
    return J
def grad(X, y, theta):
```

```
    m = len(y)
    return 1/m * X.T.dot(model(X, theta) - y)
def gradientDescentMulti(X, y, theta, alpha, num_iters):
    """
    Performs gradient descent to learn theta.
    Updates theta by taking num_iters gradient steps with learning rate alpha.

    Parameters
    ----------
    X : array_like
        The dataset of shape (m x n+1).

    y : array_like
        A vector of shape (m, ) for the values at a given data point.

    theta : array_like
        The linear regression parameters. A vector of shape (n+1, )

    alpha : float
        The learning rate for gradient descent.

    num_iters : int
        The number of iterations to run gradient descent.
```

Returns

-------

theta : array_like

The learned linear regression parameters. A vector of shape (n+1, ).

J_history : list

A python list for the values of the cost function after each iteration.

Instructions

------------

Peform a single gradient step on the parameter vector theta.

While debugging, it can be useful to print out the values of

the cost function (computeCost) and gradient here.

"""

# Initialize some useful values

m = y.shape[0] # number of training examples

# make a copy of theta, which will be updated by gradient descent

theta = theta.copy()

J_history = []

J_history=np.zeros(num_iters)

```
    for i in range (0,num_iters):
        theta=theta-alpha*grad(X,y,theta)
        J_history[i]=computeCostMulti(X,y,theta)
    return theta, J_history
```

**# initialize fitting parameters**

```
theta = np.zeros(3)
```

**# some gradient descent settings**

```
iterations = 1500
alpha = 0.01
theta, J_history = gradientDescentMulti(X ,y, theta, alpha, iterations)
print('Theta found by gradient descent: {:.4f}, {:.4f}, {:.4f}'.format(*theta))
```

## Résultat

```
]: # initialize fitting parameters
   theta = np.zeros(3)

   # some gradient descent settings
   iterations = 1500
   alpha = 0.01

   theta, J_history = gradientDescentMulti(X ,y, theta, alpha, iterations)
   print('Theta found by gradient descent: {:.4f}, {:.4f}, {:.4f}'.format(*theta
   #J_history

   Theta found by gradient descent: 340412.5630, 109370.0567, -6500.6151
```

## ***Exercise: Selection de taux d'apprentissage***

Dans cette partie de l'exercice, on va essayer différents taux d'apprentissage pour l'ensemble de données et trouver un taux d'apprentissage qui converge rapidement. On va aussi modifier le taux d'apprentissage en modifiant le code suivant et en modifiant la partie du code qui définit le taux d'apprentissage.

"""

Instructions

---

We have provided you with the following starter code that runs

gradient descent with a particular learning rate (alpha).

Your task is to first make sure that your functions - `computeCost`

and `gradientDescent` already work with  this starter code and

support multiple variables.

After that, try running gradient descent with different values of

alpha and see which one gives you the best result.

Finally, you should complete the code at the end to predict the price

of a 1650 sq-ft, 3 br house.

Hint

---

At prediction, make sure you do the same feature normalization.

"""

```
# Choose some alpha value - change this

alpha = 0.1

num_iters = 400


# init theta and run gradient descent

theta = np.zeros(3)

theta, J_history = gradientDescentMulti(X, y, theta, alpha, num_iters)


# Plot the convergence graph

pyplot.plot(np.arange(len(J_history)), J_history, lw=2)

pyplot.xlabel('Number of iterations')

pyplot.ylabel('Cost J')


# Display the gradient descent's result

print('theta computed from gradient descent: {:s}'.format(str(theta)))


price = np.dot(np.array([1.,(1650. - mu[0])/sigma[0] , (3. - mu[1])/sigma[1]]),np.array(theta) )  # You should change this

print(price)

# ================================================================


print('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): ${:.0f}'.format(price))


pyplot.figure(figsize=(15,10))

alpha=[0.01, 0.003,0.001]

for i in range(len(alpha)):

   pyplot.subplot(2,3,i+1)

   num_iters = 400
```

**# init theta and run gradient descent**

```
theta = np.zeros(3)

theta, J_history = gradientDescentMulti(X, y, theta, alpha[i], num_iters)
```

**# Plot the convergence graph**

```
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)

pyplot.xlabel('Number of iterations')

pyplot.ylabel('Cost J')
```

**# Display the gradient descent's result**

```
print('theta computed from gradient descent: {:s}'.format(str(theta)))
```

**# Estimate the price of a 1650 sq-ft, 3 br house**

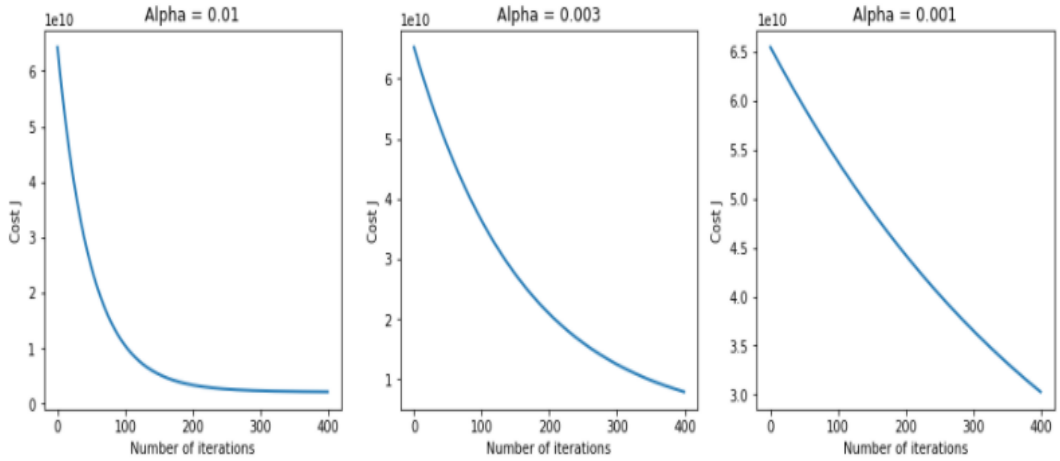
**# Recall that the first column of X is all-ones.**

**# Thus, it does not need to be normalized.**

```
price = np.dot(np.array([1.,(1650. - mu[0])/sigma[0] , (3. - mu[1])/sigma[1]]),np.array(theta) )  # You should change this

print(price)

print('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): ${:.0f}'.format(price))

print("\n")

pyplot.title('Alpha = '+str(alpha[i]))
```

## Résultat

```
289221.5473712181
Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): $289222


theta computed from gradient descent: [238067.09470609  67368.83752161  19746.42023768]
203553.10370302649
Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): $203553


theta computed from gradient descent: [112272.89290139  33254.00585532  14524.42608546]
94156.27135253145
Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): $94156
```




Donc Alpha= 0.01 est le taux d'apprentissage dans une bonne fourchette

```
# Choose some alpha value - change this
alpha = [0.1, 0.3,0.01,0.03, 0.001,0.003,0.0003,0.0001]
num_iters = 2000

# init theta and run gradient descent
for i in range(len(alpha)):
    theta = np.zeros(3)
    theta, J_history = gradientDescentMulti(X, y, theta, alpha[i], num_iters)

    # Plot the convergence graph
    pyplot.plot(np.arange(len(J_history)), J_history, lw=2,label = 'learningrate'+str(alpha[i]))
    pyplot.xlabel('Number of iterations')
```

```
pyplot.ylabel('Cost J')

pyplot.legend()

print('theta computed with from gradient descent: {:s}'.format(str(theta)))
```

# Display the gradient descent's result

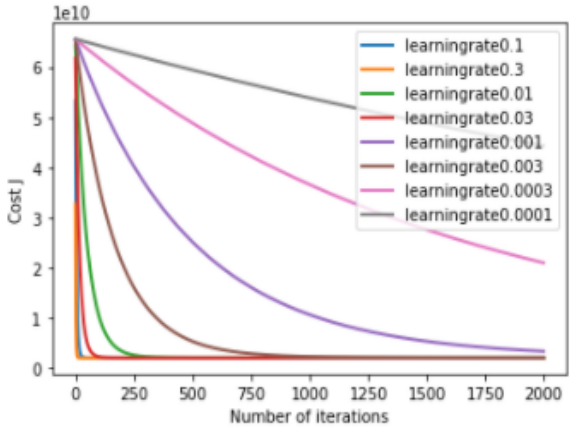
# Estimate the price of a 1650 sq-ft, 3 br house

# Recall that the first column of X is all-ones.

# Thus, it does not need to be normalized.

```
price = 0   # You should change this

print('Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): ${:.0f}'.format(price))
```

```
theta computed with from gradient descent: [340412.65957447 109447.79646964  -6578.35485416]
theta computed with from gradient descent: [340412.65957447 109447.79646964  -6578.35485416]
theta computed with from gradient descent: [340412.65894002 109439.22578243  -6569.78416695]
theta computed with from gradient descent: [340412.65957447 109447.79646948  -6578.35485399]
theta computed with from gradient descent: [294388.89339564  83125.36792731  15212.40521995]
theta computed with from gradient descent: [339576.43615572 105311.60418477  -2450.82887525]
theta computed with from gradient descent: [153607.04755151  44728.60266957  17804.15924334]
theta computed with from gradient descent: [61709.1336788  18673.5465658   8898.70264165]
Predicted price of a 1650 sq-ft, 3 br house (using gradient descent): $0
```




**Exercise: Ecrivez une methode qui permet de predire le prix d'une maison en fonction de la superfice et du nombre de piece**

**def predictnorm(superficie, pieces,theta):**

```
    theta = normalEqn(X,y)

    price = np.dot(theta.T, np.array([1., superficie, pieces]) )

    return price
```

**Appel  de fonction**

price = predictnorm(1650,3,theta)

print(f"une maison avec 500 de superficie et 10 pieces est estimée à {price} en utilisant l'equation normale")

**Résultat**

```
une maison avec 500 de superficie et 10 pieces est estimée à 293081.4643350591 en utilisant l'equation normale
```

---

## 4 Exercice: Trouvez des données et faites de la prévision en utilisant les algorithmes implémentées dans ce notebook.

**#Importation des librairies**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

%matplotlib inline

**#Chargement des donnees**

data=pd.read_csv('data/Salaire_employes.csv')

data

| | AnnesExperience | Salaire |
|---|---|---|
| 0 | 1.1 | 39343.0 |
| 1 | 1.3 | 46205.0 |
| 2 | 1.5 | 37731.0 |
| 3 | 2.0 | 43525.0 |
| 4 | 2.2 | 39891.0 |
| 5 | 2.9 | 56642.0 |
| 6 | 3.0 | 60150.0 |
| 7 | 3.2 | 54445.0 |
| 8 | 3.2 | 64445.0 |
| 9 | 3.7 | 57189.0 |
| 10 | 3.9 | 63218.0 |
| 11 | 4.0 | 55794.0 |
| 12 | 4.0 | 56957.0 |
| 13 | 4.1 | 57081.0 |
| 14 | 4.5 | 61111.0 |
| 15 | 4.9 | 67938.0 |
| 16 | 5.1 | 66029.0 |
| 17 | 5.3 | 83088.0 |
| 18 | 5.9 | 81363.0 |
| 19 | 6.0 | 93940.0 |

**#Chargement restreint des données**

data.head()

| | AnnesExperience | Salaire |
|---|---|---|
| 0 | 1.1 | 39343.0 |
| 1 | 1.3 | 46205.0 |
| 2 | 1.5 | 37731.0 |
| 3 | 2.0 | 43525.0 |
| 4 | 2.2 | 39891.0 |

---

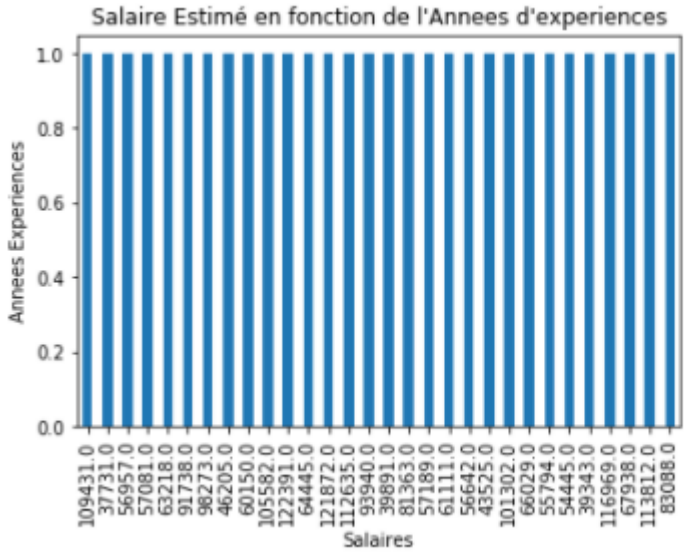
**#Description des donnes**

data.describe()

| | AnnesExperience | Salaire |
|---|---|---|
| count | 30.000000 | 30.000000 |
| mean | 5.313333 | 76003.000000 |
| std | 2.837888 | 27414.429785 |
| min | 1.100000 | 37731.000000 |
| 25% | 3.200000 | 56720.750000 |
| 50% | 4.700000 | 65237.000000 |
| 75% | 7.700000 | 100544.750000 |
| max | 10.500000 | 122391.000000 |

**#Estimation du salaire en fonction de l'année d' expérience a travers un diagramme**

```
data['Salaire'].value_counts().plot(kind='bar')

plt.title("Salaire Estimé en fonction de l'Annees d'experiences")

plt.xlabel('Salaires')

plt.ylabel('Annees Experiences')

sns.despine
```




## #Méthode qui nouspermet de predire  le salaire en fonction de l'annee d'experience

```
def predict(AnnesExperience, mu, sigma,theta):
    # init theta and run gradient descent
    n=X.shape[1]
    theta = np.zeros(n)
    theta, J_history = gradientDescentMulti(X, y, theta, alpha=0.003, num_iters=400)
    Salaire = np.dot(np.array([1.,(AnnesExperience - mu[0])/sigma[0]]),np.array(theta) )
    return (f"Pour {AnnesExperience} Annees Experiences , le Salaire estimé est %.3f"%Salaire)
```

**#Exemple deprediction du salaire pour une annee d'experience de 2.2 ans**

```
salary= predict(2.2,mu,sigma,theta)
print(salary)
```

```
Pour 2.2 Annees Experiences , le Salaire estimé est 32939.214
```