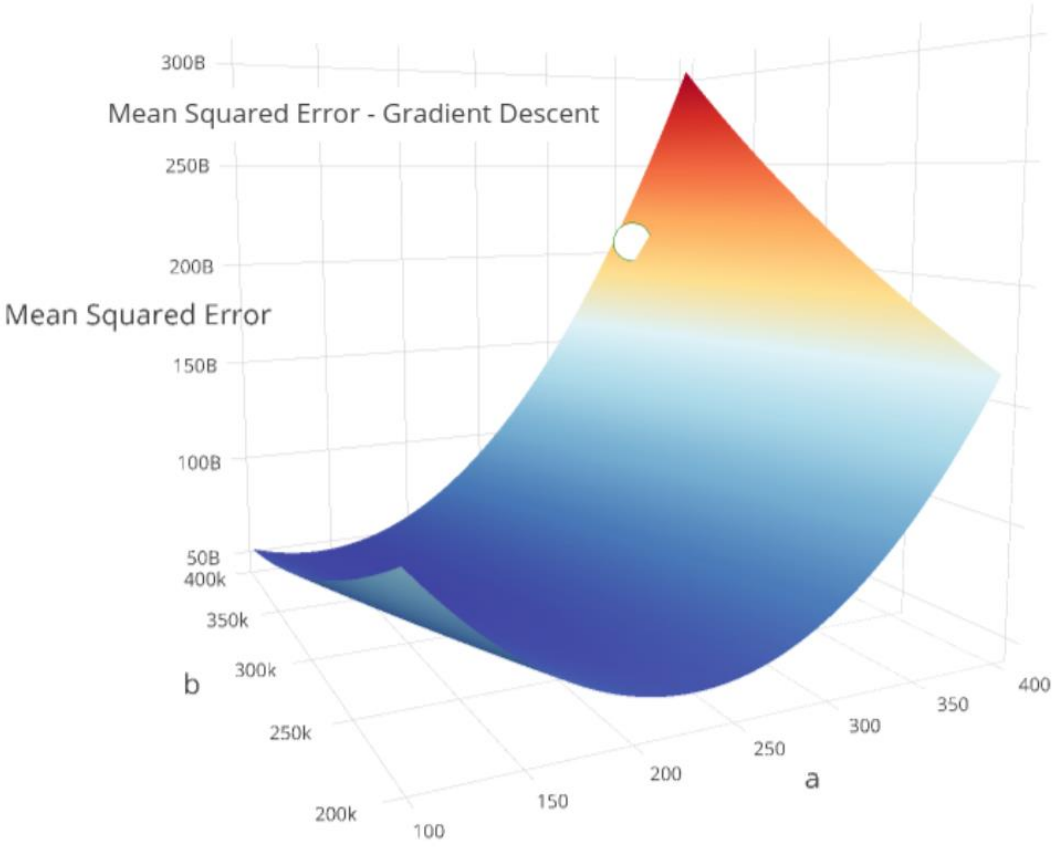


Premier Modèle I.A.

TECHNIQUES DE REGRESSIONS

CORLAY Morgan, TANGUY Franky | Compte-rendu de brief | 31/01/2022

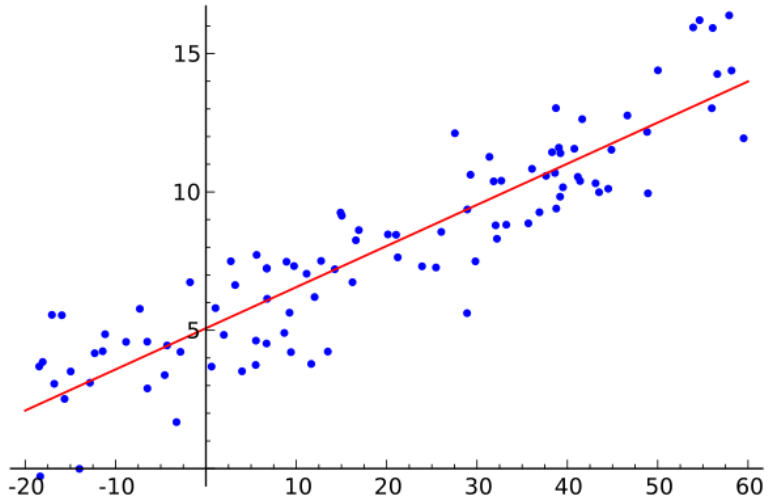
1. Rappel sur les régressions (linéaires, multiples et polynômiales

1.1 Principe

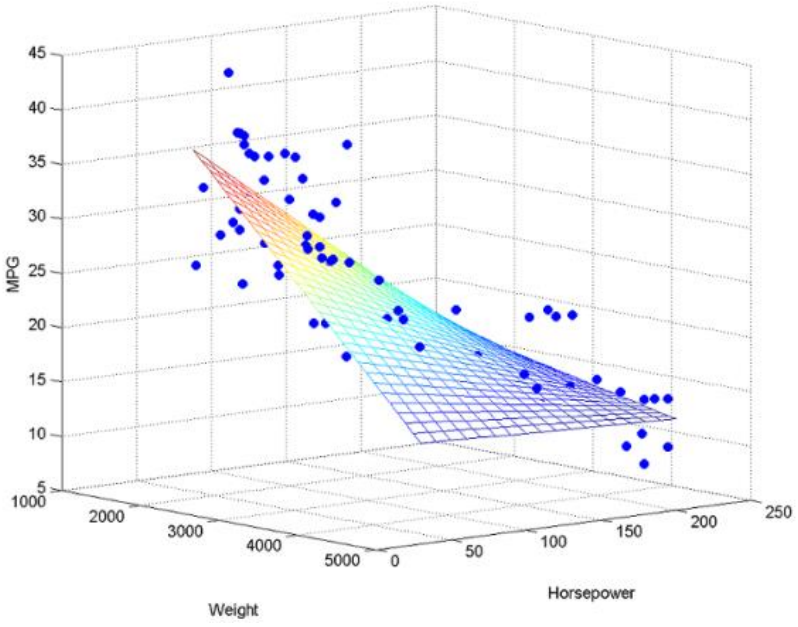
La régression est une méthode permettant de générer un modèle mathématique à partir d'un jeu de donnée. Le principe est d'obtenir une fonction mathématique correspondant au maximum à la répartition des points dans le jeu de données. On approxime une variable cible en utilisant d'autres variables qui lui sont corrélées. Les régressions sont utilisées en apprentissage automatique lorsque les variables sont quantitatives. Autrement on parle de classification.

On peut distinguer :

- La régression linéaire : dans ce cas, on cherche une approximation linéaire, donc une fonction de la forme y=a*x+b, avec x étant nos données.

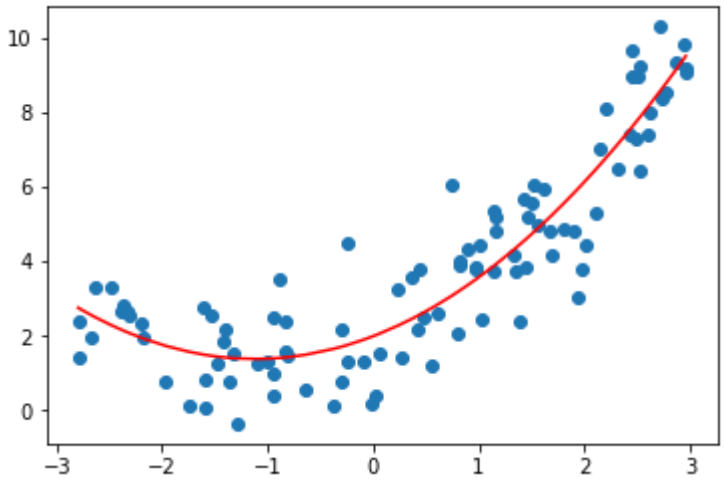


- La régression multiple : il s'agit d'une régression linéaire, à la différence qu'on a plusieurs variables en entrée. La fonction recherchée est de la forme y=a*x1+b*x2+...+n*xn, avec x1,x2,xn nos différents variables issue du dataset.



- La régression polynômiale : nous nous retrouvons dans le premier cas où on a une seule feature (une variable). La différence réside ici dans le fait que l'on ne se trouve plus dans le cas d'une variable linéaire. Le but ici est d'approcher notre jeu de données par une fonction polynômiale, de la forme :

a*x^n+b*x^(n-1)+c*x^(n-2)+....+z



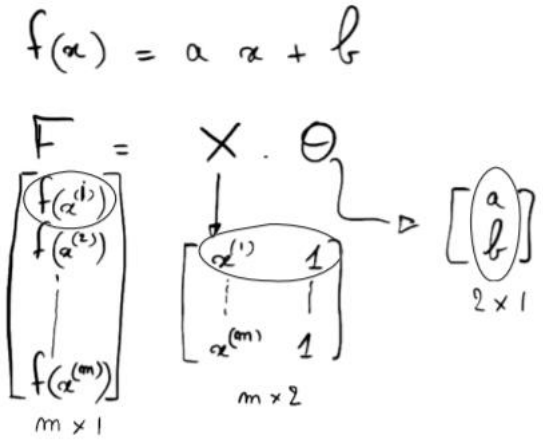
1.2 Rappels mathématiques et formes matricielles des régressions

Nous allons programmer les différentes fonctions en Python qui seront nécessaires pour faire nos régressions. Si elles sont correctement programmées, elles serviront à la fois pour la régression, linéaire, multiple et polynômiale.

Tout d'abord, nous souhaitons définir une manière de décrire notre modèle quel que soit le cas.

Tournons-nous vers le calcul matriciel, qui permettra de simplifier l'écriture des équations et facilitera le calcul informatique. Voyons comment passer des équations vers l'écriture matricielle.

Régression linéaire simple :



Régression linéaire multiple :

$$\begin{cases} y_1 = a_0 + a_1 x_{1,1} + \ldots + a_p x_{1,p} + \varepsilon_1 \\ y_2 = a_0 + a_1 x_{2,1} + \ldots + a_p x_{2,p} + \varepsilon_2 \\ \ldots \\ y_n = a_0 + a_1 x_{n,1} + \ldots + a_p x_{n,p} + \varepsilon_n \end{cases}$$

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_{1,1} & \cdots & x_{1,p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & \cdots & x_{n,p} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_p \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_n \end{pmatrix}$$

Régression polynômiale :

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0.$$

$$\begin{pmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \ldots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \ldots & x_1 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \ldots & x_n & 1 \end{pmatrix} \begin{pmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}$$

2. Explication de chaque fonction (par la méthode normale)

Nous allons définir les différentes fonctions nécessaire aux régressions par apprentissage.

2.1 Fonction model

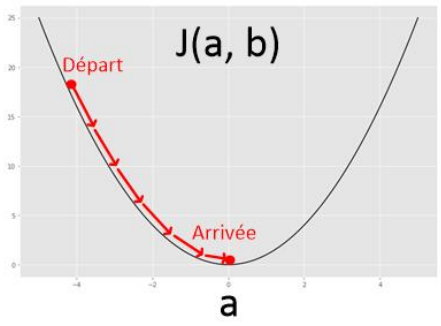
Si F est notre modèle, x est l'ensemble des variables et theta la liste des coefficients qui leurs sont associés, on peut simplifier les équations vues précédemment :

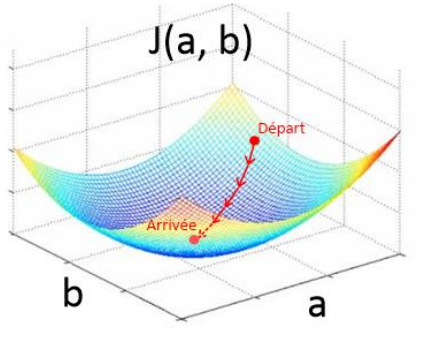
F=x*theta (cette équation matricielle est valable quel que soit le type de régression, il faut juste penser à redimensionner la matrice x selon les cas)

```
#Définition du modèle
def model(x,theta):
    f=np.dot(x,theta)
    return f
```

2.2 Fonction coût

Cette fonction va nous permettre de calculer le coût de la fonction, c'est-à-dire une estimation de l'erreur entre le modèle que l'on crée et celui qu'on veut approcher. Le but est de trouver le minimum de cette fonction, qui de part son expression est de forme convexe.



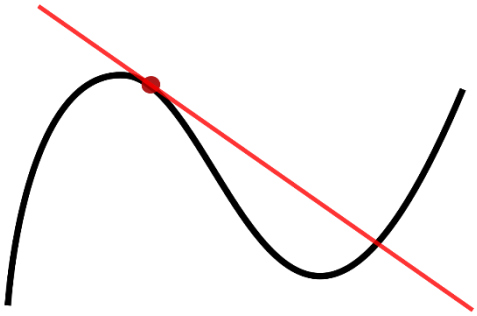


On peut ainsi visualiser l'apprentissage de notre programme, comme nous le verrons par la suite.

```
#Fonction coût
def fonction_cout(x,y,theta):
    m = len(y)
    return (1/(2*m)) * np.sum((model(x, theta) - y)**2)
```

2.3 Calcul du gradient

Petits rappels concernant le gradient



Soit f une fonction quelconque, on peut commencer par calculer sa dérivée.

En mathématiques, la dérivée d'une fonction d'une variable réelle mesure l'ampleur du changement de la valeur de la fonction (valeur de sortie) par rapport à un petit changement de son argument (valeur d'entrée). On peut dire que la dérivée en un point de la fonction représente la valeur de la pente en ce point.

Le gradient est une généralisation de la dérivée à n-dimension. Par définition, c'est un vecteur contenant toutes les dérivées partielles de la fonction dans notre espace de dimension n. Il s'écrit de la manière suivante :

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

Dans notre cas, on calcule le gradient comme ci-dessous :

```
#Calcul du gradient
def grad(x,y,theta):
    dJ=(1/len(y))*(x.T.dot(model(x,theta)-y))

    return(dJ)
```

2.4 Descente de gradient

L'algorithme de descente de gradient va fonctionner de la manière suivante :

- On initialise notre modèle avec des paramètres a, b, c,... aléatoires (notre vecteur theta).
- On calcule le coût.
- On calcule le gradient.
- Le gradient calculé permet de mettre à jour le vecteur theta de manière à minimiser la fonction coût.

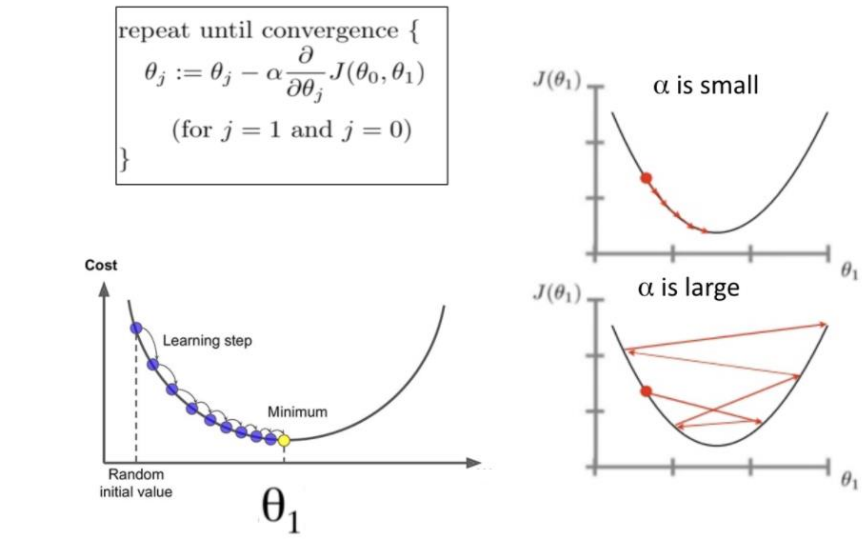
On répète ce cycle autant de fois que nécessaire.

```
-  #Descente de gradient
-  def descente_gradient(x,y,theta,alpha,n_iterations):
-      cout = np.zeros(n_iterations)
-      for i in range (0,n_iterations):
```

```
-           f=model(x,theta);
-           # if i%10==0:
-           #     plt.figure()
-           #     plt.plot(x, model(x,theta), lw=0.5)
-           cout[i] = fonction_cout(x, y, theta)
-           theta = theta - alpha * grad(x,y,theta)
-       return(theta, cout)
-
-   n_iterations = 40
-   learning_rate = 0.0001
```

Il est important de noter que l'on souhaite converger vers le minimum de la fonction coût, en prenant en compte le nombre d'itération, et en définissant un pas (le learning rate ici noté alpha). Ce pas agit sur la sensibilité de notre algorithme en terme de mise à jour de theta et ne doit pas être choisi de manière irréfléchie.
En effet on peut distinguer trois cas, selon que la valeur d'alpha soit trop faible, trop élevée ou dans une tranche raisonnable.

Learning rate (α)

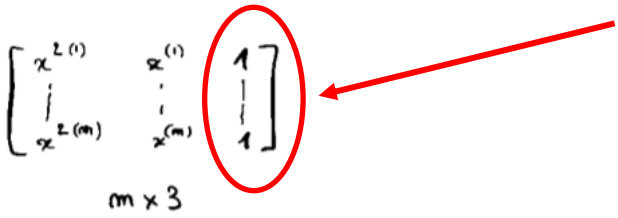


Si le learning rate est trop faible, l'apprentissage sera insuffisant, il faudra alors augmenter le nombre d'itérations, ce qui rendra l'exécution lente.
Si le learning rate est trop élevé, la descente de gradient ne pourra pas se faire, le pas trop grand entrainant des oscillations autour du minimum de la fonction coût sans jamais l'atteindre. Ce paramètre joue aussi sur la précision du modèle fourni.
Il est nécessaire de faire un compromis entre le learning rate et le nombre d'itération, de manière à converger vers une solution assez rapidement.
Pour lancer l'algorithme, nous exécutons le code suivant :

```
theta_final, cost_history = descente_gradient(x, y, theta,
learning_rate, n_iterations)
```

3. Présentation des résultats (codé à la main)

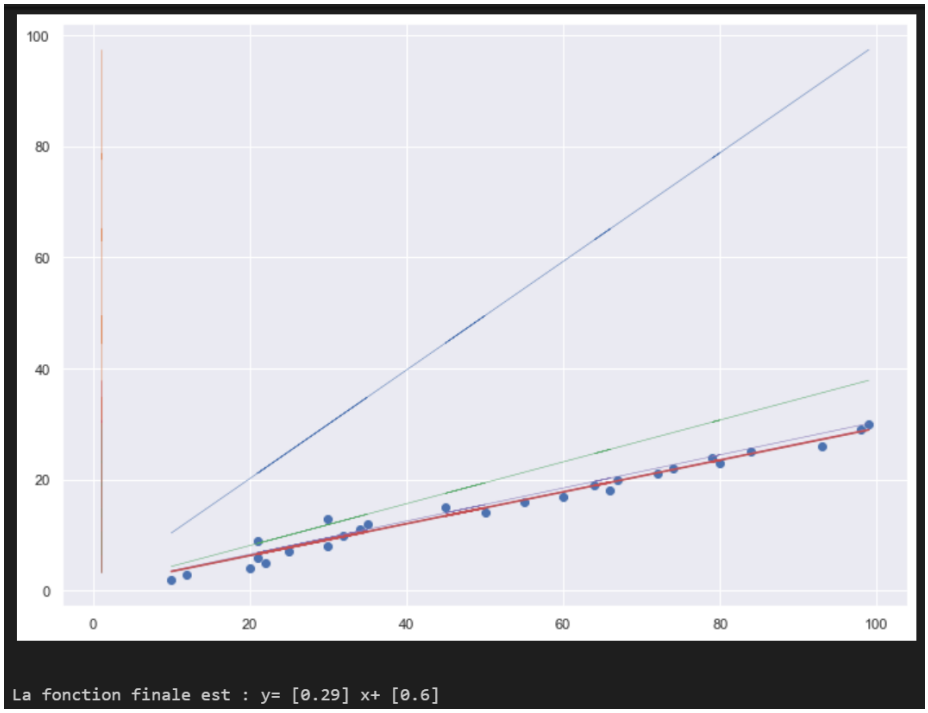
Nous allons appliquer nos fonctions pour faire des régressions sur différents jeux de données. Il est nécessaire d'expliquer qu'il a un prétraitement à faire afin de faire fonctionner les calculs matriciels correctement.
Sans rentrer dans les détails, il nous faut rajouter une colonne de 1 à la matrice x, de manière à obtenir correctement l'ordonnée à l'origine, le coefficient non relié aux variables. La matrice x sera dans tous les cas comme-ci :



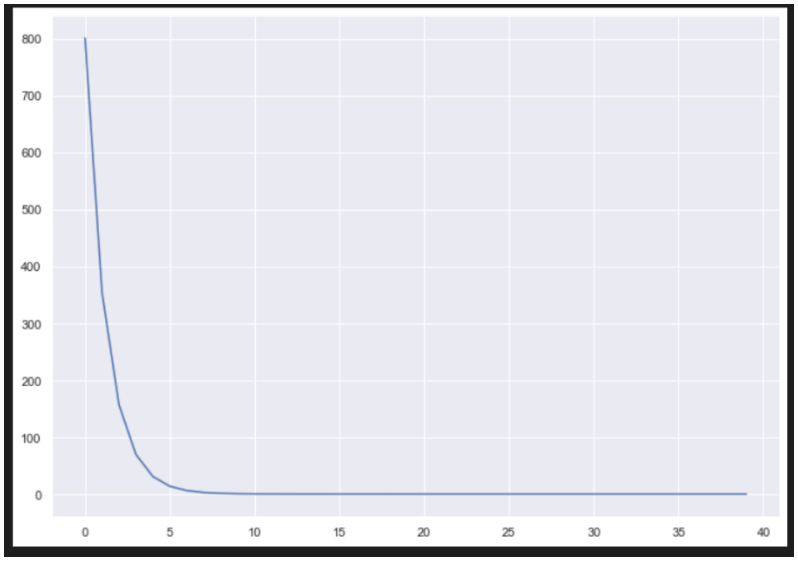
3.1 Régression linéaire

Notre premier jeu de donnée présente une feature (entrée) et une target (sortie). Nous sommes donc dans le cas d'une régression linéaire simple.

```
# Création d'un vecteur prédictions qui contient les prédictions de
notre modele final
predictions = model(x, theta_final)

# Affiche les résultats de prédictions (en rouge) par rapport a notre
Dataset (en bleu)
plt.scatter(x0, y)
plt.plot(x0, predictions, c='r')
plt.show()

# Affichage de la fonction prédite
print("La fonction finale est :
y=",theta_final[0].round(2),"x+",theta_final[1].round(2))

#Affichage de la fonction coût en fonction du nombtre d'itérations
plt.plot(range(n_iterations), cost_history)
plt.show()
```



Nous affichons volontairement le résultat de la droite toutes les 10 itérations pour observer la convergence vers le modèle final.

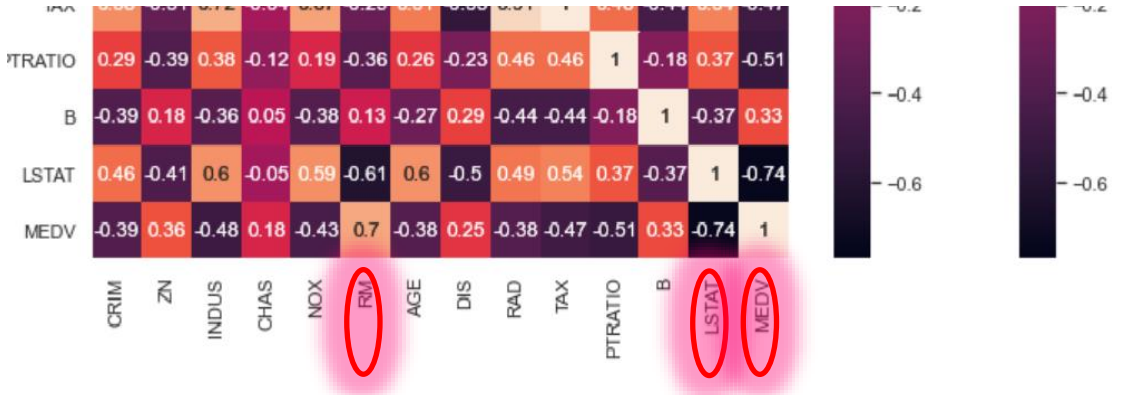


On observe la fonction coût se stabiliser à un plateau : l'algorithme a bien appris.

3.2 Régression linéaire multiple

Dans la dataframe suivante, nous avons une target et 13 features. Nous décidons d'observer les différentes corrélations entre les variables. En effet, faire de la régression linéaire sur des variables non corrélées avec la target n'a pas vraiment de sens.

```
correlation_matrix = data2.corr().round(2)
sns.heatmap(data=correlation_matrix, annot=True)
```




```
(506, 3) (506,)
[[6.575 4.98  1.   ]
 [6.421 9.14  1.   ]
 [7.185 4.03  1.   ]
 ...
 [6.976 5.64  1.   ]
 [6.794 6.48  1.   ]
 [6.03  7.88  1.   ]]
```

De cette grille, nous remarquons que deux features en particulier ont une corrélation prononcée avec la target : RM et LSTAT. Ce sont donc les features que nous allons utiliser pour notre régression linéaire multiple. De plus, le fait de se cantonner à deux features nous permettra de faire une visualisation dans un espace 3D.

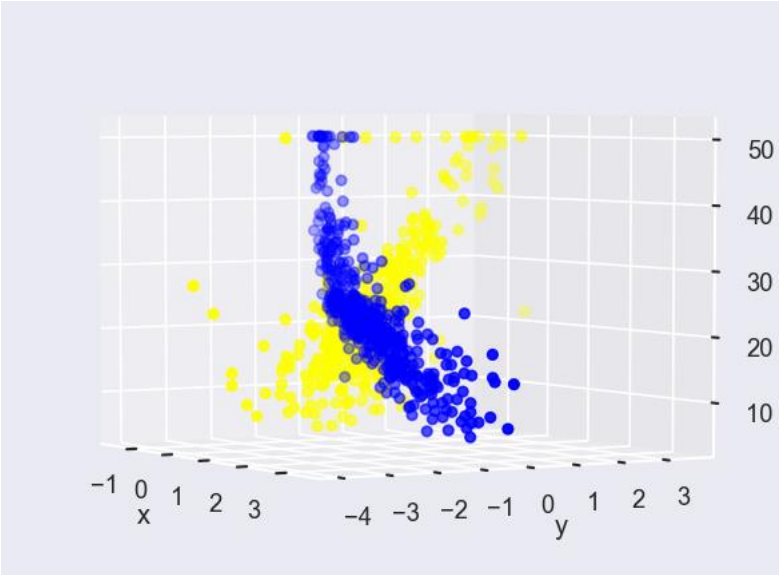
```
scaler = StandardScaler()
print(scaler.fit(x))
StandardScaler()

x=scaler.transform(x)
print(x)
print('x =',x)

x1=x[:,0]
x2=x[:,1]

y = data2.iloc[:,-1].values
y = y.reshape((len(y),1))
```

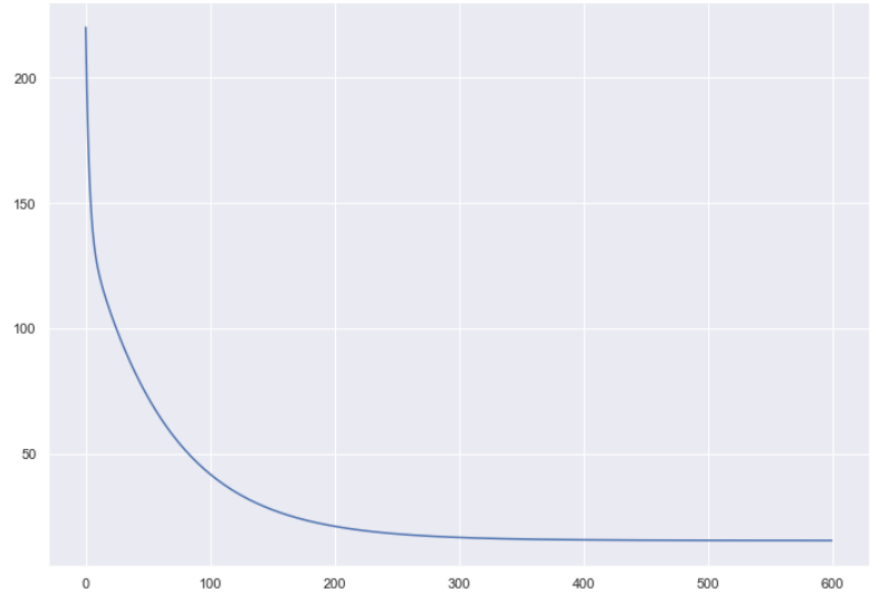
Comme nous avons deux targets, nous décidons d'utiliser la fonction StandardScaler(), afin de standardiser nos données. En effet, nous ne voulons pas qu'une des targets ait un poids plus important dans la descente de gradient.

```
(506, 2) (27, 1)
StandardScaler()
[[ 0.41367189 -1.0755623 ]
 [ 0.19427445 -0.49243937]
 [ 1.28271368 -1.2087274 ]
 ...
 [ 0.98496002 -0.98304761]
 [ 0.72567214 -0.86530163]
 [-0.36276709 -0.66905833]]
x = [[ 0.41367189 -1.0755623 ]
 [ 0.19427445 -0.49243937]
 [ 1.28271368 -1.2087274 ]
 ...
 [ 0.98496002 -0.98304761]
 [ 0.72567214 -0.86530163]
 [-0.36276709 -0.66905833]]
```

Nous utilisons nos fonctions créées précédemment, et nous obtenons les résultats suivants :

```
Theta original : [[0.06827237]
 [0.28111175]
 [0.50725529]]
La fonction finale est : y=  [4.71] x1 +  [-0.65] x2 +  [1.13359771]
Theta final : [[ 4.70609885]
 [-0.6500917 ]
 [ 1.13359771]]
```






```
MSE =  30.573846956992913
```

3.3 Régression polynômiale

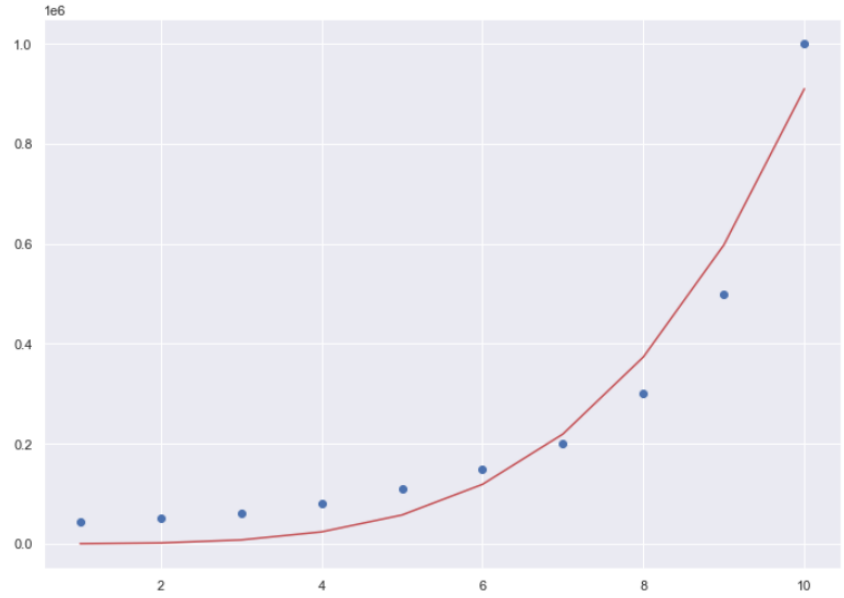
Les données pour la régression polynômiale présentent un salaire en fonction de la position de l'employé dans l'entreprise.

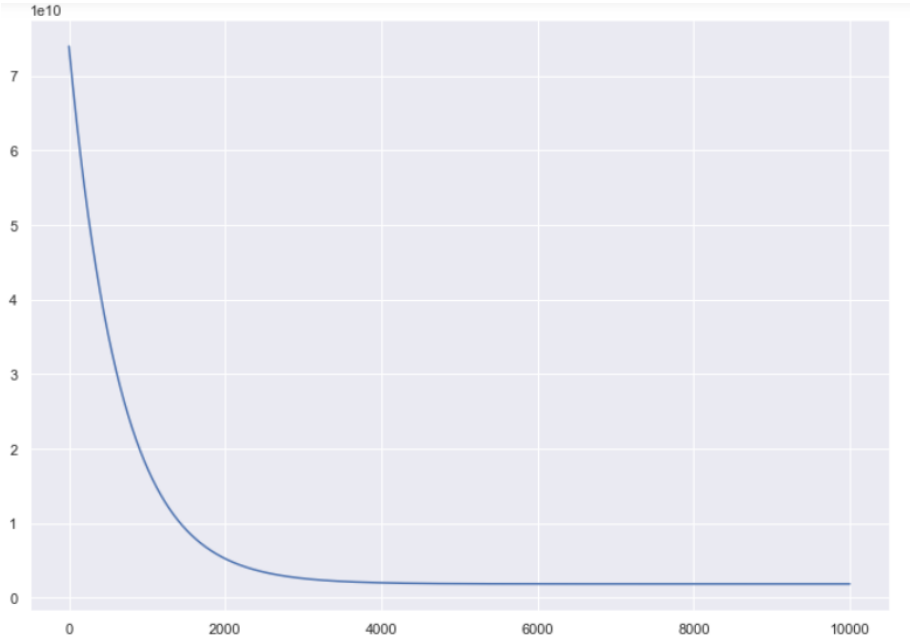
```
data3=pd.read_csv('./Data_Reg/Position_Salaries.csv')
print(data3)
```

0.4s

	Position	Level	Salary
0	Project Analyste	1	45000
1	Ingenieur	2	50000
2	Senior Consultant	3	60000
3	Manager	4	80000
4	Country Manager	5	110000
5	Gouverneur	6	150000
6	Associate	7	200000
7	Commercial	8	300000
8	C-level	9	500000
9	PDG	10	1000000

```
(10, 5) (10, 1)
[[  45000]
 [  50000]
 [  60000]
 [  80000]
 [ 110000]
 [ 150000]
 [ 200000]
 [ 300000]
 [ 500000]
 [1000000]]
Theta original : [[-1.5322392 ]
 [-1.11795429]
 [ 0.66780035]
 [-0.20444314]
 [ 0.67307999]]
Theta final : [[ 9.01119370e+01]
 [ 8.43860060e+00]
 [ 1.72361249e+00]
 [-7.17323214e-02]
 [ 6.95250708e-01]]
```



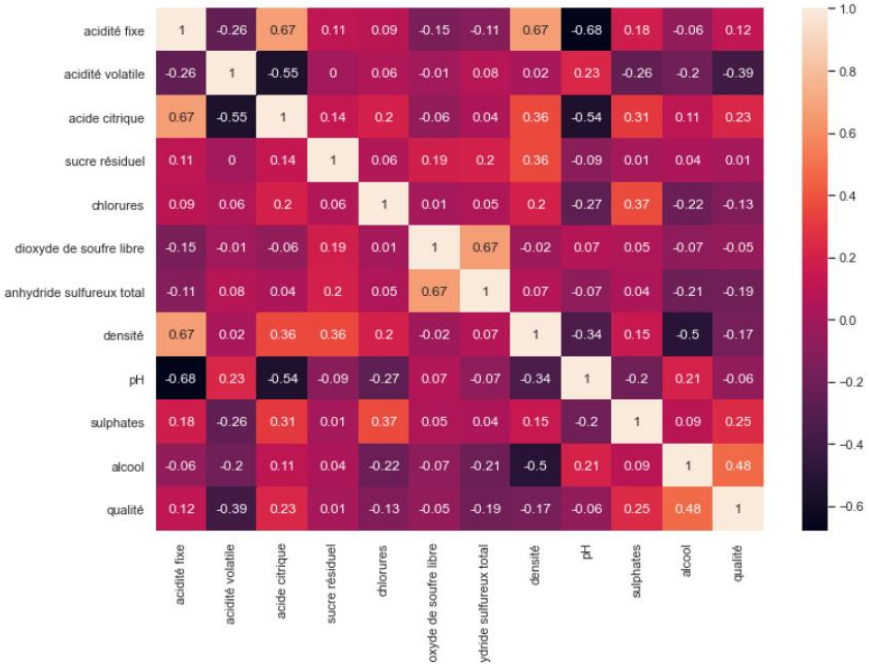


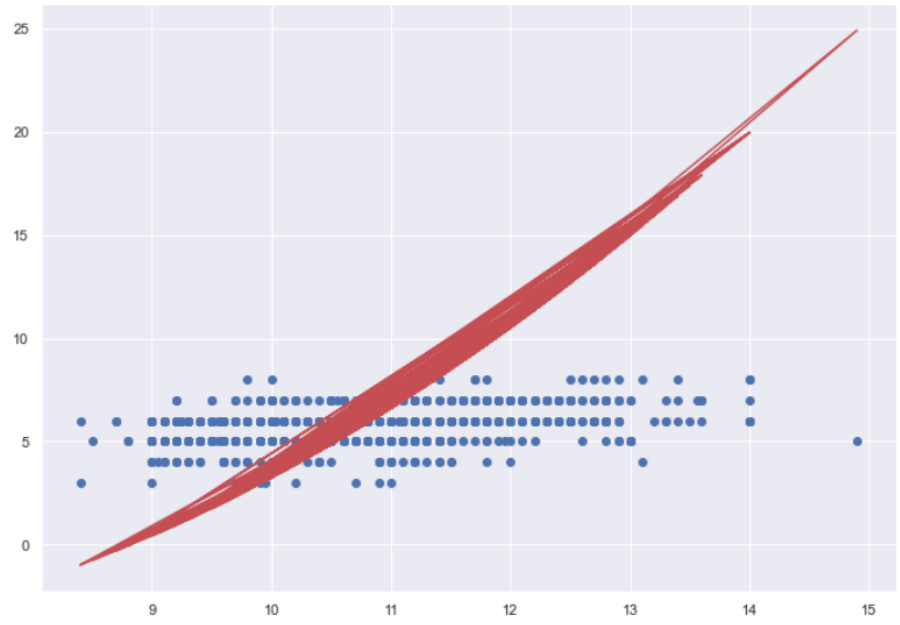

```
: mse=mean_squared_error(y,model(x,theta_opt))
  print('MSE =',mse)
  R_2=1-(sum(y-model(x,theta_opt))**2)/(sum((y-np.mean(y))**2))
  print('R² =',R_2)
```

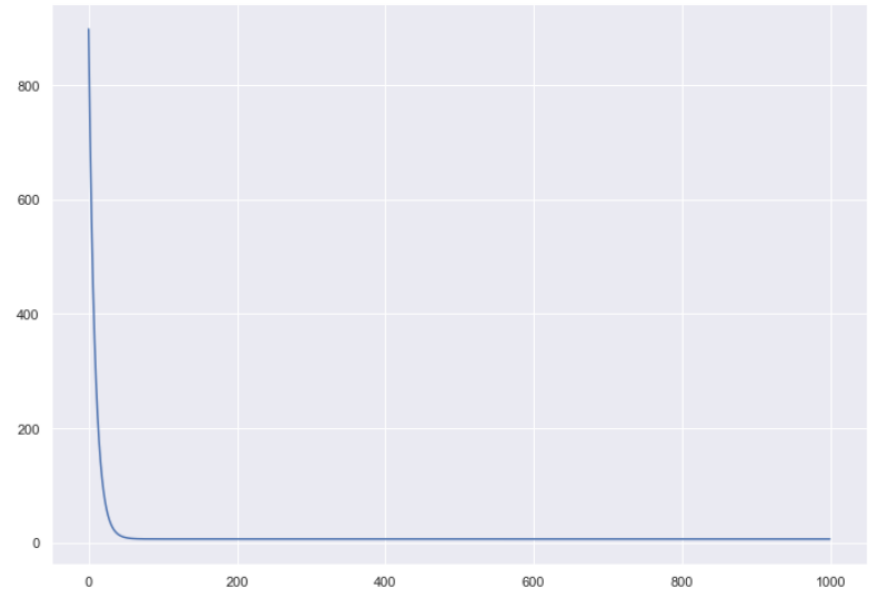
```
MSE = 3748001887.2022605
R² = [0.95710962]
```

3.4 Régression polynômiale sur un jeu de donnée non adapté

```
      acidité fixe  acidité volatile  acide citrique  sucre résiduel  \
0              7.4             0.700            0.00             1.9
1              7.8             0.880            0.00             2.6
2              7.8             0.760            0.04             2.3
3             11.2             0.280            0.56             1.9
4              7.4             0.700            0.00             1.9
...            ...               ...             ...             ...
1594           6.2             0.600            0.08             2.0
1595           5.9             0.550            0.10             2.2
1596           6.3             0.510            0.13             2.3
1597           5.9             0.645            0.12             2.0
1598           6.0             0.310            0.47             3.6

      chlorures  dioxyde de soufre libre  anhydride sulfureux total  densité  \
0         0.076                     11.0                       34.0  0.99780
1         0.098                     25.0                       67.0  0.99680
2         0.092                     15.0                       54.0  0.99700
3         0.075                     17.0                       60.0  0.99800
4         0.076                     11.0                       34.0  0.99780
...         ...                      ...                        ...      ...
1594      0.090                     32.0                       44.0  0.99490
1595      0.062                     39.0                       51.0  0.99512
1596      0.076                     29.0                       40.0  0.99574
1597      0.075                     32.0                       44.0  0.99547
1598      0.067                     18.0                       42.0  0.99549

        pH  sulphates  alcool  qualité
0     3.51       0.56     9.4        5
1     3.20       0.68     9.8        5
2     3.26       0.65     9.8        5
3     3.16       0.58     9.8        6
4     3.51       0.56     9.4        5
...    ...        ...     ...      ...
1594  3.45       0.58    10.5        5
1595  3.52       0.76    11.2        6
1596  3.42       0.75    11.0        6
1597  3.57       0.71    10.2        5
```







Même si nous observons une convergence de la descente de gradient, on ne peut qu'être insatisfait des résultats obtenus. Les données étant regroupées par cluster, on pouvait s'y attendre. Ces données pourraient faire l'objet d'une classification, mais pas d'une régression.

Evaluation des résultats

4. Présentation des résultats avec le module Scikit-Learn

4.1 Régression linéaire simple

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.2, random_state = 0)
```

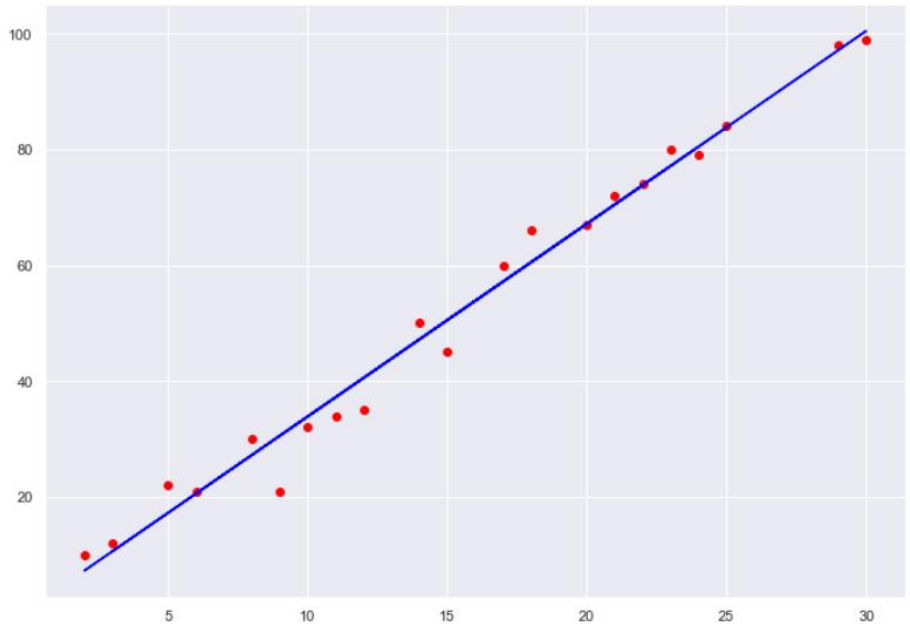
```
from sklearn.linear_model import LinearRegression
model_linreg = LinearRegression()
model_linreg.fit(X_train, y_train)
```

```
LinearRegression()
```

```
y_pred = model_linreg.predict(X_test)
a=model_linreg.coef_
b=model_linreg.intercept_
print(a,b)
```

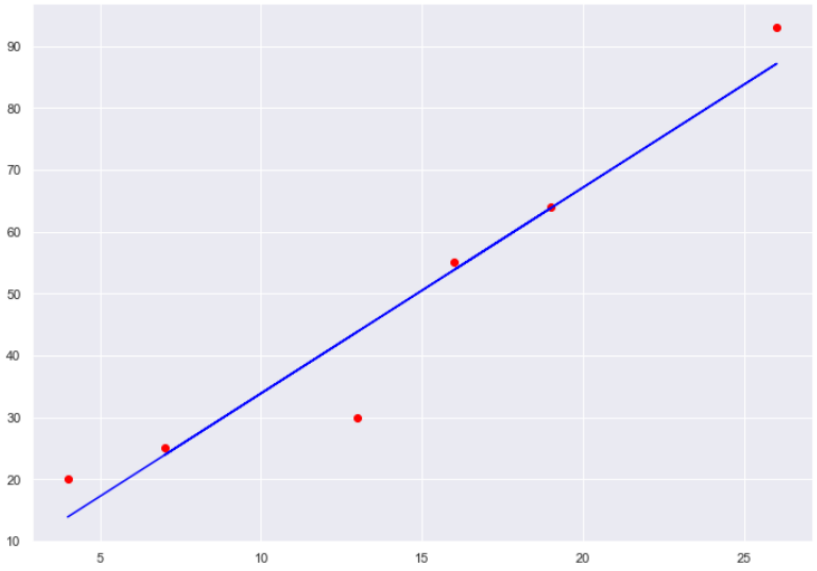
```
[3.33070866] 0.5643044619422639
```

```
#Visualising the training set result
plt.scatter(X_train, y_train, color = 'red')
plt.plot(X_train, model_linreg.predict(X_train), color = 'blue')
plt.show
```



```
plt.scatter(X_test, y_test, color = 'red')
plt.plot(X_test, model_linreg.predict(X_test), color = 'blue')
plt.show
sc=r2_score(y_test, model_linreg.predict(X_test))
print("Score= ",sc)
```

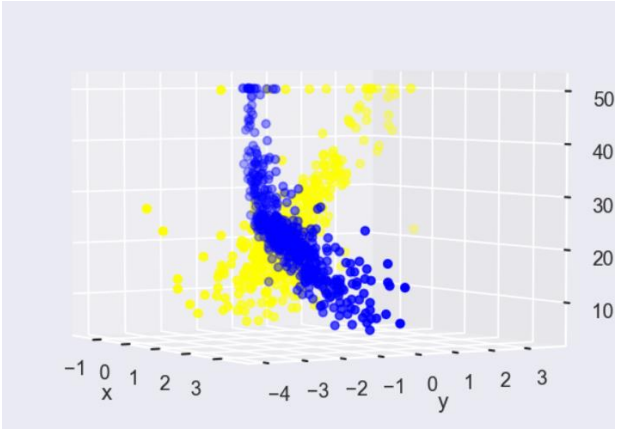
```
core=  0.9328868520347855
```




```
Score=  0.9328868520347855
mean_squared_error = 44.37111207555747
mean_absolute_error = 4.705161854768154
```

4.2 Régression linéaire multiple

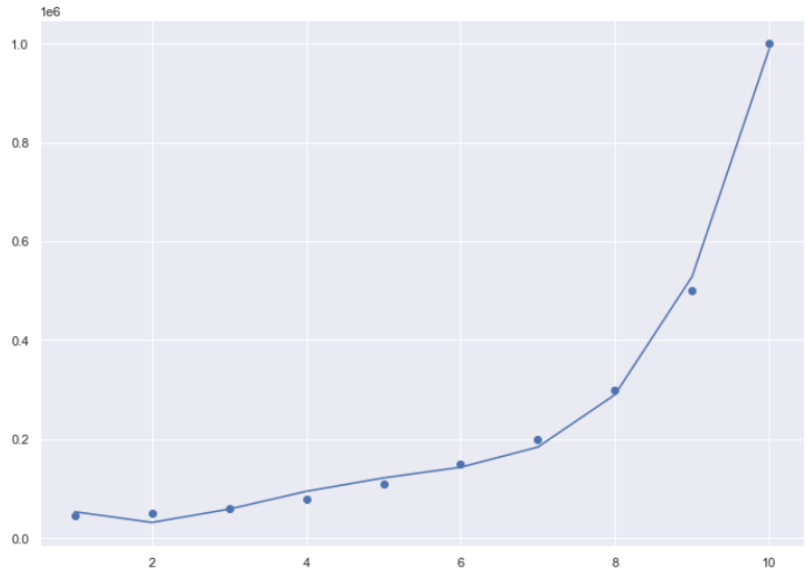
```
(506, 2) (506, 1)
x = [[6.575 4.98 ]
 [6.421 9.14 ]
 [7.185 4.03 ]
 ...
 [6.976 5.64 ]
 [6.794 6.48 ]
 [6.03  7.88 ]]
Precision : 0.6618625964841894
Prediction : [[-1.91747748]
 [ 6.99076186]
 [15.89900119]
 [24.80724052]
 [33.71547985]]
```



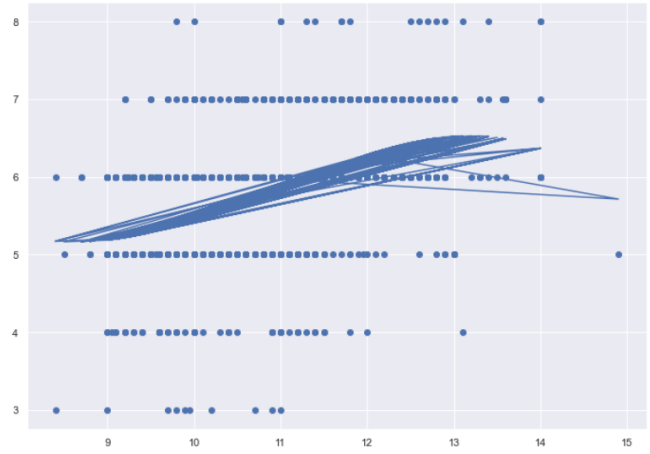
4.3 Régression polynômiale

```
from sklearn.preprocessing import PolynomialFeatures

nb_degree = 4
polynomial_features = PolynomialFeatures(degree = nb_degree)
x_poly = polynomial_features.fit_transform(x)

model = LinearRegression()
model.fit(x_poly, y)

y_pred = model.predict(x_poly)

rmse = np.sqrt(mean_squared_error(y,y_pred))
r2 = r2_score(y,y_pred)

print('RMSE: ', rmse)
print('R2: ', r2)


plt.figure()
plt.scatter(x,y)
plt.plot(x,y_pred)
plt.show
```

```
RMSE:  14503.234909626824
R2:  0.9973922891706614

<function matplotlib.pyplot.show(*args, **kw)>
```



Reprenons notre dataset sur les vins, non adapté à une régression.



Comme il était à prévoir, sklearn ne nous aidera pas à réaliser une régression polynômiale sur notre dataset, les données étant regroupées par classes.

5. Comparaison avec la méthode normale

Les résultats pour la régression linéaire simple sont similaires avec ou sans scikit-learn, de même pour la régression linéaire multiple.

En revanche, scikit-learn nous donne de meilleurs résultats sur la régression polynômiale. En effet, le learning rate est très sensible dès lors qu'on augmente le degré du polynôme, et le fait de ne pas avoir à s'en occuper nous enlève cette difficulté.

Conclusion

Ce projet a été pour nous une première approche du machine learning. Avant d'utiliser des modules et leurs fonctions, il nous a fallu comprendre les fondamentaux. La régression est une méthode idéale, car plutôt simple à programmer à la main. La programmation de la régression linéaire simple est celle qui nous a pris le plus de temps, il nous a fallu comprendre mathématiquement le problème, et surtout manipuler les données en pré-traitement. Pour la régression multiple, la problématique était de savoir le nombre de features à prendre en compte et lesquelles. La régression polynômiale nous a sensibilisé à la problématique du learning rate et des erreurs pouvant être associées (overflow…). Pour finir, le dataset sur le vin nous a montré que la régression n'était pas adaptée. Nous nous trouverions plutôt sur un problème de classification.
À l'issue de ce projet, nous nous sentons plus à l'aise avec la logique derrière le machine learning :

- Prétraitement des données : mise en forme des matrices, standardisation.
- Apprentissage : choix du learning rate et du nombre d'itérations.
- Visualisation et interprétation des résultats : techniques d'évaluation des modèles.

Nous avons acquis les bases nécessaires pour poursuivre le cursus et une compréhension suffisante pour étudier des modèles plus complexes.