# Premier modèle IA

31 janvier 2022

**Auteurs :**

Fabio Cassiano



# Table des matières

# 1 Rappel sur la régression linéaire

La régression linéaire est une méthode statistique qui a pour objectif de trouver une relation entre une variable cible (*target*) à partir d'une variable dite descriptive. La régression linéaire peut-être appliqué sur différents types de modèles qui vont être abordés dans les sous-section ci-dessous.

## 1.1 Modèle simple

Le modèle linéaire simple, est un modèle basé sur une fonction affine. Cette fonction a pour équation mathématique :

$$f(x) = ax + b$$

Dans cette équation $f(x)$ représente la valeur cible, soit celle qu'il faut prédire. La valeur $x$ correspond à la variable descriptive à partir de laquelle on pourra obtenir $f(x)$. Les valeurs a et b correspondent aux coefficients, qui représentent respectivement la pente et l'ordonnée à l'origine.

## 1.2 Modèle multiple

Le modèle linéaire multiple est une extension du modèle linéaire simple, qui permet la prédiction du variable cible à partir de plusieurs variables descriptives. Ce modèle a pour équation :

$$f(x) = ax_1 + bx_2 + c$$

Tout comme pour l'équation précédente $f(x)$ représente la valeur cible, soit celle qu'il faut prédire. Les valeur $x_1$ jusqu'à $x_n$ représentent aux différentes variables descriptives de notre jeu de données.

## 1.3 Modèle polynomiale

Concernant le modèle polynomiale, il correspond à une équation à une inconnue, avec un polynôme de degré $n$. Mathématiquement cela se traduit de la manière suivante :

$$f(x) = ax^n + bx^{n-1} + c$$

Par exemple pour un polynomial de degré 2 l'équation est :

$$f(x) = ax^2 + bx + c$$

# 2 Fonction sous python

## 2.1 Définition de la fonction *model()*

Les différents modèles présentés précédemment peuvent être retranscrit sous python. Pour facilité leur modélisation on peut les écrire modèles sous forme matricielle, ce qui ce résumera entre un produit matricielle entre deux matrices.

### 2.1.1 Modèle simple

L'écriture matricielle du modèle linéaire simple est donc la suivante :

$$F = X \cdot \theta$$

avec,

$$X = \begin{bmatrix} x^{(1)} & 1 \\ \vdots & \vdots \\ x^{(n)} & 1 \end{bmatrix}$$

et,

$$\theta = \begin{bmatrix} a \\ b \end{bmatrix}$$

où $X$ est la matrice qui contient la variable descriptive $x$, sous forme de colonne. Une colonne composé uniquement de 1 est également inclut dans $X$, ce qui correspond au multiplicateur du coefficient $b$. Enfin la matrice $\theta$ regroupe les coefficients $a$ et $b$.

### 2.1.2 Modèle multiple

Pour le modèle multiple, la forme matricielle est la même que pour le modèle simple. Il suffit d'adapter la matrice $X$ en ajoutant les différentes features que l'on souhaite étudié, et ajouter à la matrice $\theta$ le nombre de coefficient correspondant. Ce qui nous donne :

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & 1 \\ \vdots & \vdots & \vdots \\ x_1^{(n)} & x_2^{(n)} & 1 \end{bmatrix}$$

### 2.1.3 Modèle polynomiale

De même pour le modèle polynomiale, il suffit d'adapter la matrice $X$ et $\theta$. Ce qui nous donne :

$$X = \begin{bmatrix} x^{2(1)} & x^{(1)} & 1 \\ \vdots & \vdots & \vdots \\ x^{2(n)} & x^{(n)} & 1 \end{bmatrix}$$

### 2.1.4 Code python

Au niveau code cela se traduit facilement, il suffit de mettre en place une fonction *model()*, avec deux paramètres d'entrées, nos matrices $X$ et $\theta$.

```
1 # Création de la fonction modélisant le modèle linéaire F
2 def model(X, theta):
3     return X.dot(theta)
```

## 2.2 Définition de la fonction *fonction_ cout()*

La fonction de coût permet de mesurer les erreurs entre le modèle calculé et le jeu de données. La formule mathématiquement de cette fonction, pour le linéaire simple, s'écrit de la manière suivante :

$$J(a,b) = \frac{1}{2m}\sum_{i=1}^{m}(ax + b - y)^2$$

On peut également l'écrire sous forme matricielle, ce qui la généralise pour les autres modèles présenté précédemment. Ce qui donne :

$$J(\theta) = \frac{1}{2m}\sum(X.\theta - y)^2$$

### 2.2.1 Code python

Pour implémenter cette fonction on définit une fonction *fonction_ cout()*, qui prend comme paramètres les matrices $X$, $Y$ (la *"target"*), et $\theta$. La fonction fera directement appel à la fonction précédente *model()*.

```
# Création de la fonction permettant l'estimation de la fonction de
    coût
def fonction_cout(X, Y, theta):
    m = len(X) # taille de la matrice X
    return (1/(2*m))* np.sum(np.power((model(X, theta) - Y), 2))
```

## 2.3 Définition de la fonction *gradient()*

Le gradient permet de caractérisé la variabilité d'une fonction en un point donné. Pour ce faire on calcule les dérivées partielles. Le gradient de notre fonction coût correspond donc aux dérivées partielle selon $a$ et selon $b$. Ce qui se traduit par :

$$\frac{\delta J(a,b)}{\delta a} = \frac{1}{m}\sum(ax + b - y)$$
$$\frac{\delta J(a,b)}{\delta b} = \frac{1}{m}\sum(ax + b - y)$$

De même ces équations peuvent s'écrire sous forme matricielle :

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

Où $X^T$ correspond à la transposé de $X$.

### 2.3.1 Code python

Le calcul de gradient est implémenté sous une fonction *gradient()*, prenant en paramètres d'entrées $X$, $Y$, et $\theta$. La fonction fera directement appel à la fonction précédente *model()*.

```
# Création de la fonction permettant l'estimation des gradient
def gradient(X, Y, theta):
    m = len(X)
    return (1/m) * X.T.dot((model(X, theta) - Y))
```

## 2.4 Définition de la fonction de descente de

Maintenant que les principales fonctions principal ont été définit on peut les regrouper dans une fonction *descente_gradient()*. Cette fonction a pour objectif de faire évoluer notre modèle afin de minimiser son erreur. Cette fonction prend donc en paramètres nos matrices $X$, $Y$ et $\theta$, ainsi qu'un paramètre *alpha* et un paramètre *n_iteration*. Le paramètre *alapha* correspond au pas d'apprentissage (*learning rate*) de notre modèle, c'est le facteur qui joue un rôle dans la ré-estimation du gradient. Le dernier paramètre correspond au nombre d'itération, c'est à dire le nombre de fois que nous souhaitons ré-estimer notre modèle. Dans la définition de notre fonction nous avons établis des valeurs par défaut pour ces deux paramètres, qui sont respectivement de $1e^{-3}$ et 30.

Dans cette fonction il a été décidé de commencer par initialiser deux listes *F_plot* et *J* qui stockeront respectivement tous les modèles calculés par la suite et leurs valeurs de coût.

A l'étape suivante on ré-estime un nouveau $\theta$, que l'on utilise pour calculer le nouveau modèle que l'on stocke dans *F_plot*. On calcul enfin l'erreur du modèle grâce à *fonction_cout()*, que l'on stocke dans *J*. On ré-itère ensuite ces calculs autant de fois que le nombre d'itération définit. Et pour finir on renvoi les variables *J*, *F_plot*, ainsi que les derniers coefficient estimé (la matrice $\theta$).

```
1  def descente_gradient(X, Y, theta, alpha=1e-3, n_iterations=30):
2      # Variable permetant de stocker tout les modèles pour les repré
       senter facilement
3      F_plot = []
4      J = [] # Variable pour stocker le coût
5
6      for i in range(n_iterations):
7          # Re-estimation de thêta
8          theta = theta - alpha*gradient(X, Y, theta)
9          # Calcul du model avec le nouveau thêta
10         F_plot.append(model(X, theta))
11         # Calcul du coût du modèle
12         J.append(fonction_cout(X, Y, theta))
13
14     return J, theta, F_plot
```

## 2.5 Définition de la fonction *coeff_determination()*

Le coefficient de détermination ($R^2$) permet d'évaluer le modèle estimé. Sa formule mathématique est la suivante :

$$R^2 = 1 - \frac{\sum(y - F)^2}{\sum(y - \bar{y})^2}$$

Pour notre implémentassion en python la fonction prendra en paramètre notre matrice $Y$, ainsi que la prédiction de notre modèle.

```
1  def coeff_determination(y, F):
2      return 1 - np.sum((y - F)**2) / np.sum((y - y.mean())**2)
```

# 3 Résultats des fonctions implémentés manuellement

## 3.1 Modèle simple

Le premier jeu de données (ou *DataSet*) sur lequel nous allons tester nos fonctions, est le DataSet "*reg_simple.csv*" qui regroupe les notes obtenus par des étudiants selon le nombres d'heure de révision effectué. Visuellement il est possible de constater que le DataSet a une tendance linéaire.

Après avoir préparer nos matrices $X$, $Y$ et $\theta$, on appelle notre fonction *descente_gradient()*, en gardant l'*alpha* et le *n_iteration* par défaut. Il est maintenant possible de tracer l'évolution de notre fonction (voir Fig. 1).

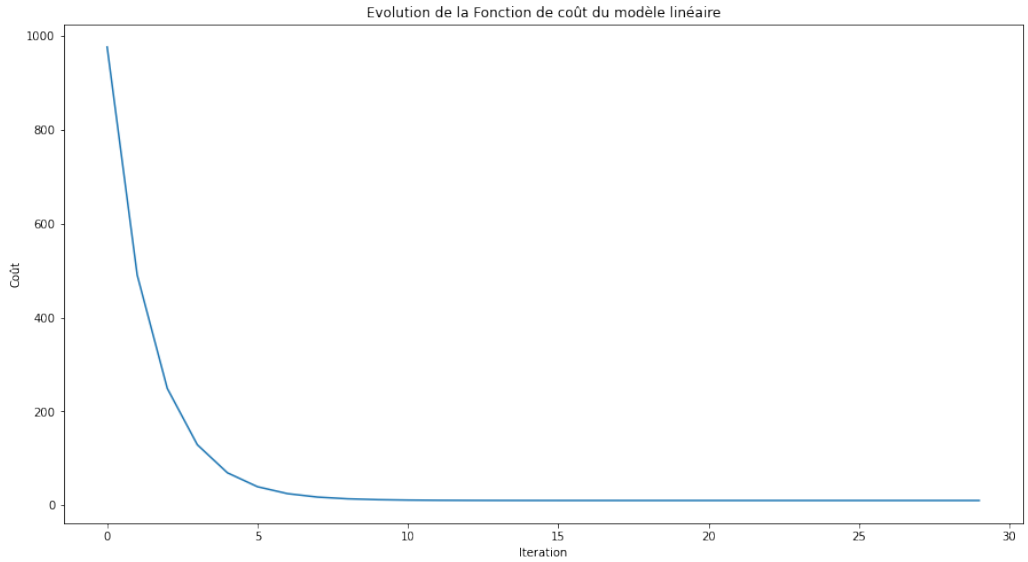



FIGURE 1 – Évolution des résultats de la fonctions de coût au fur et à mesure des itérations.

Comme on peut le voir sur ce graphique, notre coût atteint un plateau autour de la 10ème itération, ce qui signifie que notre modèle n'est plus optimisé au-delà. Il aurait donc été possible de ne réaliser que 10 itérations.

Il est également possible de représenter l'évolution de notre modèle (voir Fig. 2), grâce à notre varible *F_plot*. Ce qui nous permet de voir l'amélioration de notre modèle au fur et à mesure des itérations.

Notre modèle peut être évalué par le coefficient de détermination, pour ce faire on utilise la fonction *coeff_determination*. Pour ce premier modèle notre résultat $R^2$ est de **0.97**, ce qui est un très bon score car on se rapproche de 1.

## 3.2 Modèle multiple

Nous pouvons également utilisé nos fonctions dans le but d'estimer un modèle multiple. Pour ce faire le DataSet *boston_house_price.csv* est utilisé. Dans un premier temps on sélectionne les deux variables descriptive les plus corrélées avec notre target. les deux

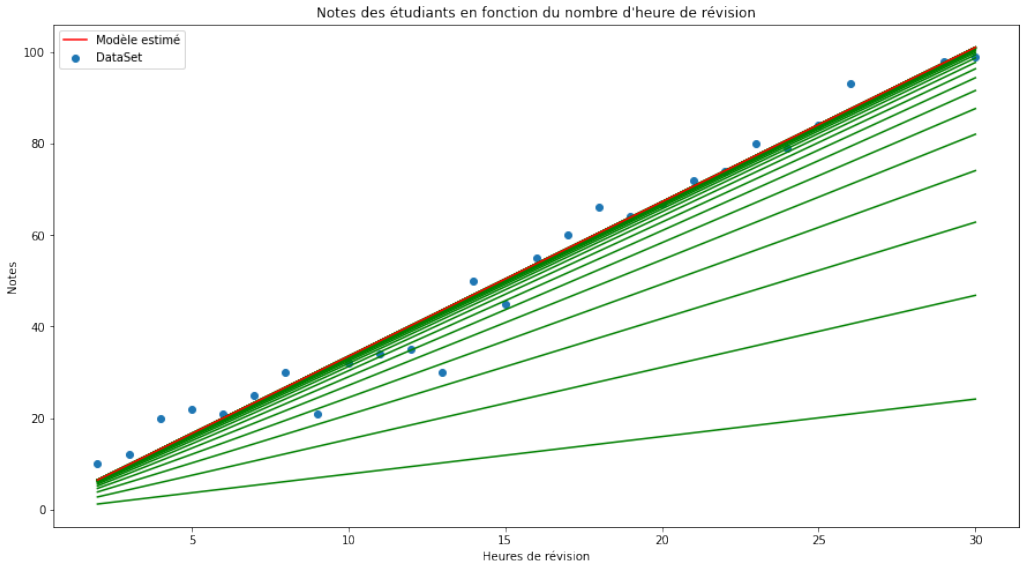



FIGURE 2 – Représentation de l'évolution du modèle au fur et à mesure des itérations.

variables sélectionnées sont *LSTAT* et *RM* (voir Fig. 3).

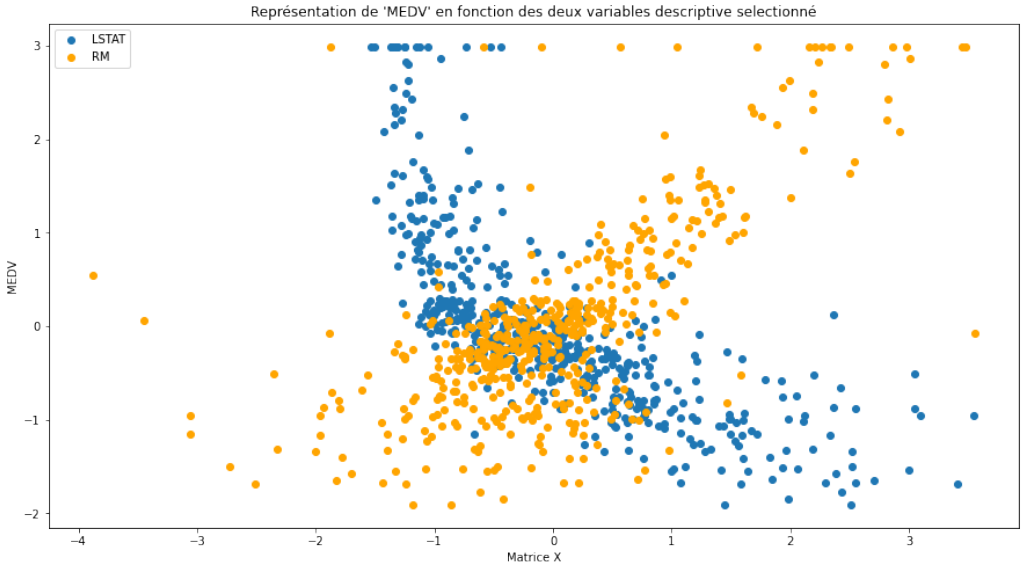



FIGURE 3 – Représentation de notre target en fonction des deux variables sélectionnés *LSTAT* et *RM*

Tout comme précédemment on prépare nos matrices et on utilise les fonctions implémentés, on trace ensuite notre graphique de fonction de coût (voir Fig. 4), en adaptant cette fois-ci nos paramètres *alpha* et *n_iteration*.

Il est possible de constater par cette représentation que nous avons commencer à atteindre un plateau autour de la 1000ème itération. Après évaluation de notre modèle avec le

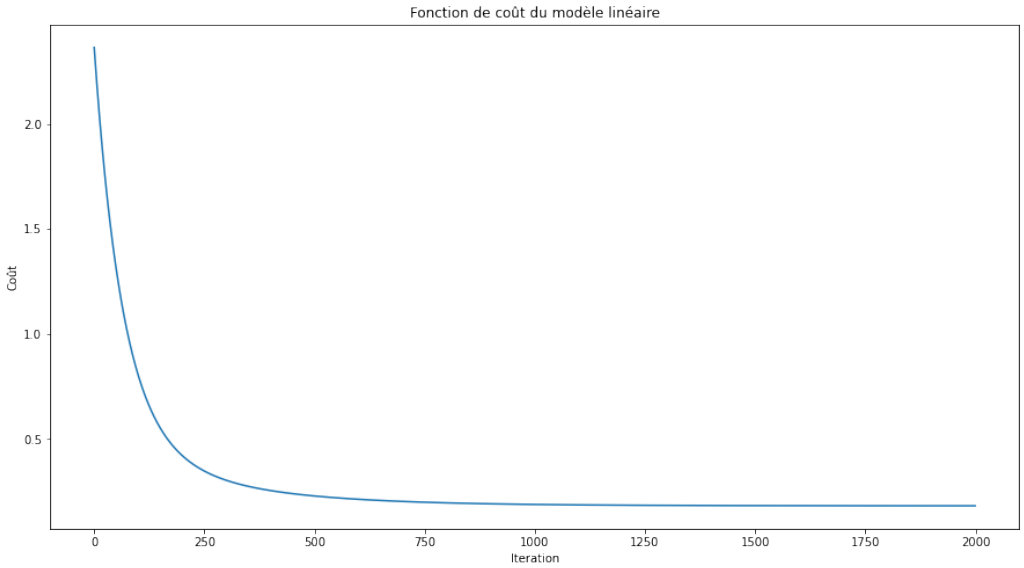



FIGURE 4 – Représentation de l'évolution des résultats de la fonctions de coût au fur et à mesure des itérations

coefficient de détermination on obtient un $R^2$ est de **0.64**. Au vu de ce résultat, on peut envisager que notre modèle n'est pas très bien adapter à ce jeu de données.

## 3.3 Modèle polynomiale

### 3.3.1 Jeu de données salaire

Pour un modèle polynomiale nous pouvons également utilisé nos fonctions. Ici le DataSet utilisé est *Position_Salaries.csv*. Comme pour les utilisations précédentes on prépare nos matrices, et l'on adapte nos paramètres *alpha* et *n_iteration*. On représente ensuite la fonction de coût (voir Fig. 5).

Pour ce DataSet on constate que l'on commence à atteindre un plateau autour de la 1500ème itération. Nous avons également tracer l'évolution du modèle en représentant les modèles obtenues toutes les 500 itérations (voir Fig. 6). L'évaluation du modèle avec le coefficient de détermination nous a donné un $R^2$ est de **0.81**. On peut considérer que notre modèle n'est pas le plus optimal, mais son résultat n'est pas non plus mauvais.

### 3.3.2 Jeu de données vin

On réalise les mêmes étapes que les précédentes sur le jeu de données *qualite-vin-rouge.csv*. On réalise une corrélation pour identifier la variable déscriptive la plus corrélé avec notre target (*"qualité"*). On établis nos différentes matrices, et on lance l'estimation de notre modèle. Il est possible de constater sur la courbe de fonction coût (voir Fig. 7), qu'un plateau est atteint autour de la 500ème itération.

Pour l'évaluation de modèle on a eu pour le coefficient de détermination **0.23**, et pour l'erreur quadratique moyenne **0.5**. Au vu de ces valeurs notre modèle n'est pas fiable.

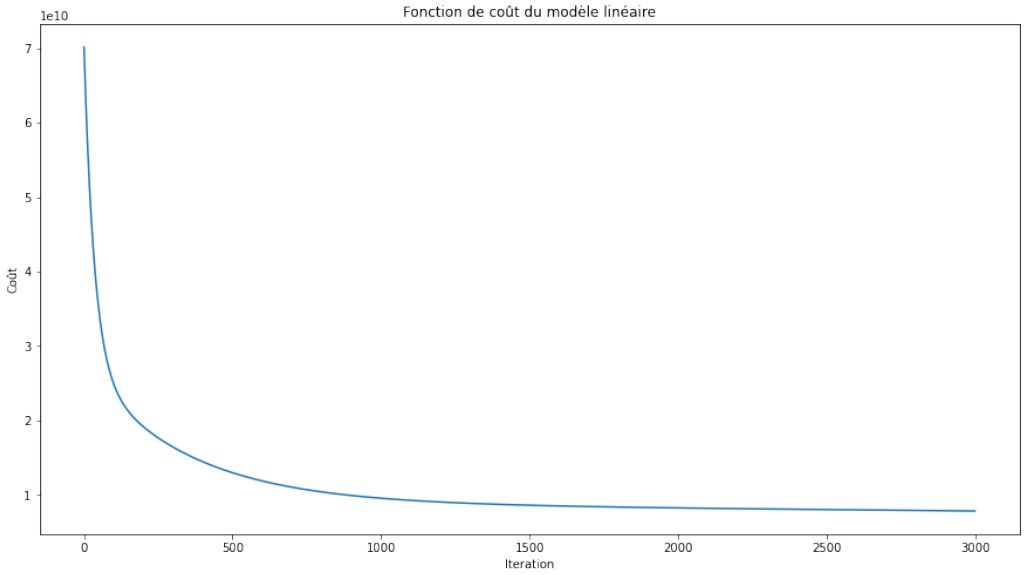



FIGURE 5 – Représentation de l'évolution des résultats de la fonctions de coût au fur et à mesure des itérations.

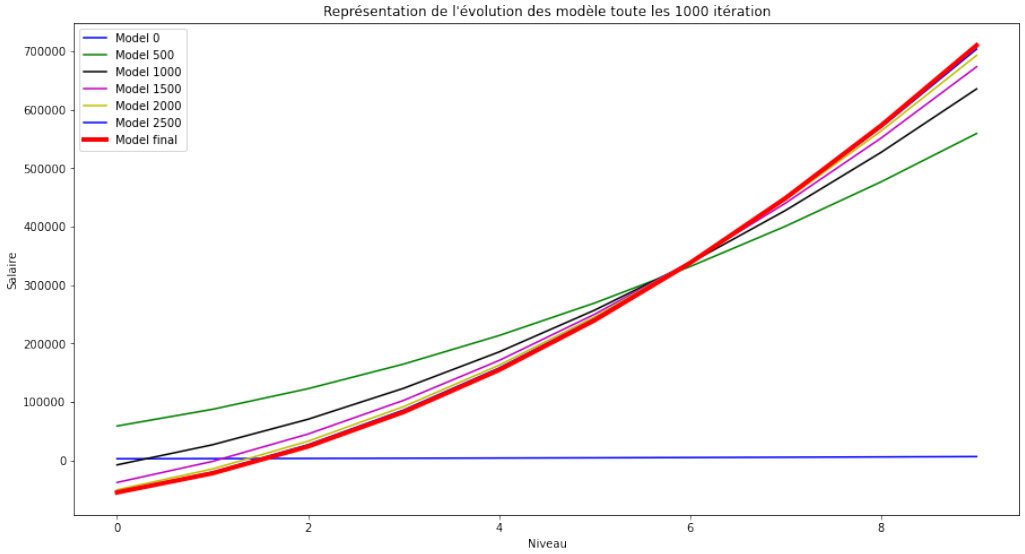



FIGURE 6 – Représentation de l'évolution du modèle toutes les 500 itérations.

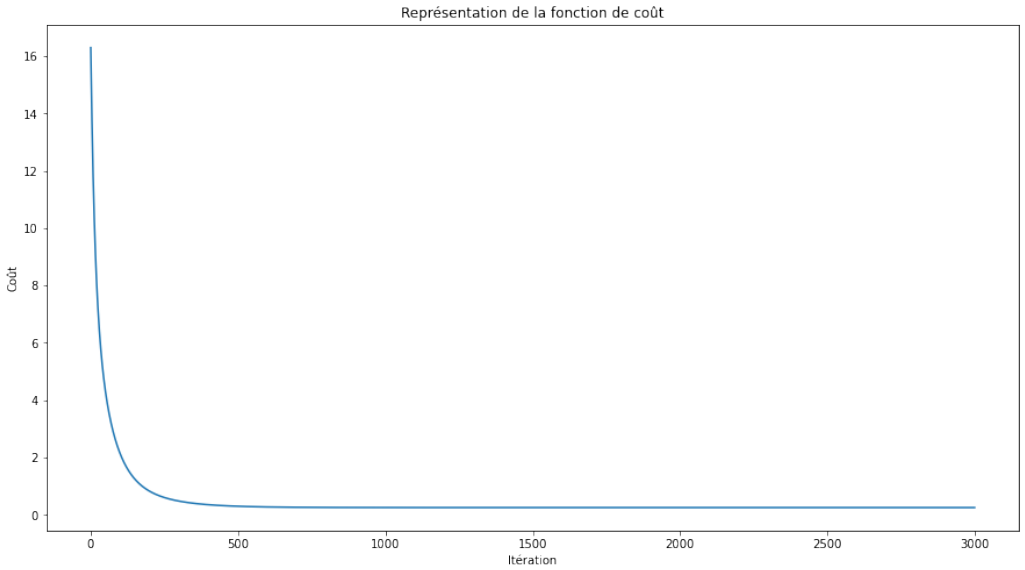



FIGURE 7 – Représentation de l'évolution des résultats de la fonctions de coût au fur et à mesure des itérations.

En représentant notre modèle sur les données (voir Fig. 8),on constate que notre jeu de données correspond à de la catégorisation. Notre approche n'est donc pas approprié à ce type de données, il faudrait peut-être réfléchir à une autre méthode.

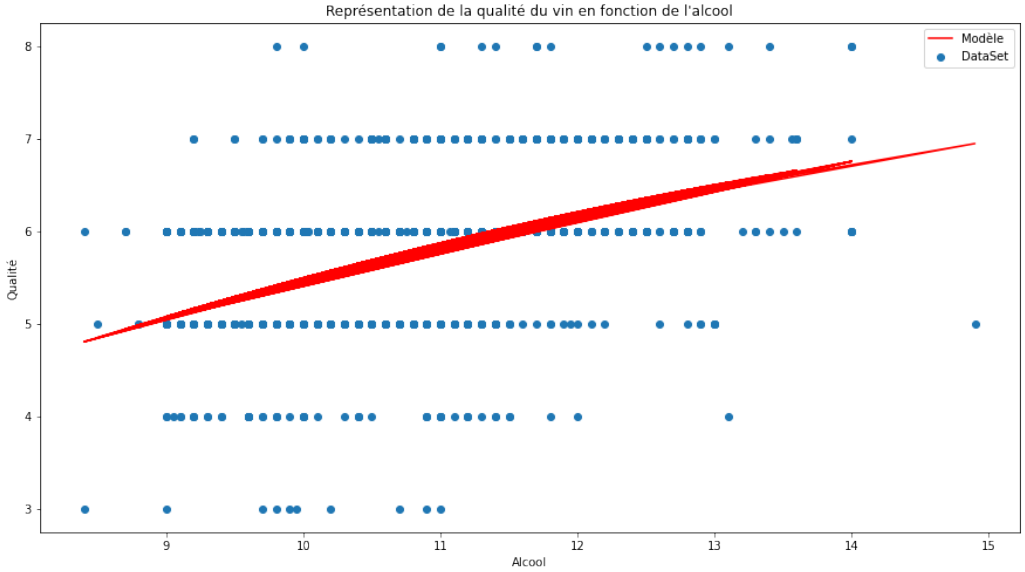
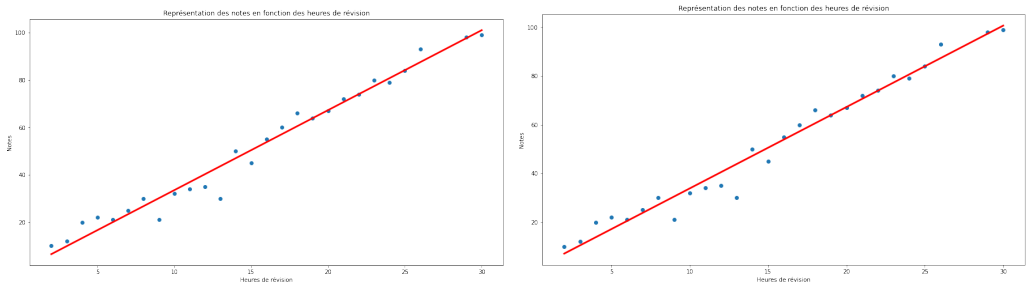



FIGURE 8 – Représentation du modèle estimé.

# 4 Comparaison des résultats

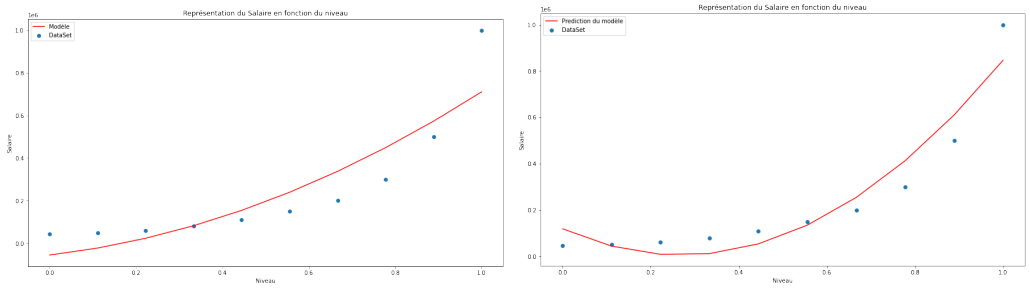
## 4.1 Modèle simple




(a) Fonction implémentée (b) ScikitLearn

FIGURE 9 – Comparaison des modèles obtenu par la fonction de régression Linéaire implémenter manuellement (a) et celle de de ScikitLearn (b)

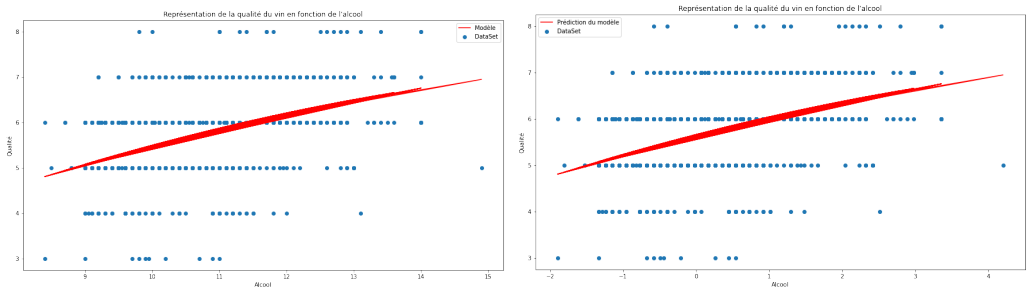
## 4.2 Modèle polynomiale - *Position_Salaries.csv*




(a) Fonction implémentée (b) ScikitLearn

FIGURE 10 – Comparaison des modèles obtenu par la fonction de régression Linéaire implémenter manuellement (a) et celle de de ScikitLearn (b)

## 4.3 Modèle polynomiale - *qualite-vin-rouge.csv*




(a) Fonction implémentée (b) ScikitLearn

FIGURE 11 – Comparaison des modèles obtenu par la fonction de régression Linéaire implémenter manuellement (a) et celle de de ScikitLearn (b)

### 4.4 Récapitulatif des évaluations des modèles

Le tableau ci-dessous regroupe les résultats obtenus lors des évaluations des différents modèles avec le coefficient de détermination et l'erreur quadratique moyenne. Les valeurs obtenus nous montre qu'il n'y a que très peu de différence entre l'utilisation des fonctions de Scikit Learn et celles que nous avons implémenté manuellement. Sauf pour le modèle polynomiale basé sur le DataSet *Position_Salaries.csv*, où l'on observe un meilleur $R^2$ pour la fonction de Scikit Learn. Cela pourrait peut-être s'expliquer par le fait que les paramètres utilisé dans notre fonction ne correspondait pas à ceux utilisé par défaut dans Scikit Learn.

| | **Fonction manuelles** | | **Fonction Scikit Learn** | |
|---|---|---|---|---|
| | $R^2$ | **MSE** | $R^2$ | **MSE** |
| **Modèle linéaire simple** | 0.97 | - | 0.97 | - |
| **Modèle linéaire multiple** | 0.64 | 0.36 | 0.64 | 0.36 |
| **Modèle polynomiale 1** | 0.81 | $1.6e^{10}$ | 0.92 | $6.8e^{9}$ |
| **Modèle polynomiale 2** | 0.23 | 0.50 | 0.23 | 0.50 |

TABLE 1 – Tableau de comparaison des valeurs de $R^2$ et de MSE obtenue par les modèles estimés par les fonctions implémenter manuellement, et les modèles obtenus par l'utilisation de Scikit Learn

Pour conclure cette comparaison, on pourrait dire que le fait d'implémenter une fonction soit même nous offre une meilleur maîtrise de son utilisation. Cependant, on peut considérer que les fonctions Scikit Learn ont sûrement était plus optimisé que les nôtres. Avec des résultats aussi similaire autant utiliser les fonctions déjà existantes qui nous permettant même dans certain cas d'obtenir de meilleurs résultats.

## 5 Conclusion

En conclusion ce projet m'a permis de revoir des notions que je connaissais déjà, je n'ai donc rien appris de nouveaux, cependant cela m'a permit de consolider mes connaissances. Aucunes difficultés à signaler.