Premier modèle IA

Régression linéaire simple, multiple, polynomiale

Nathan Faudeil, Bertrand Borel

Avant-propos :

Le projet réalisé au mois de janvier et explicité ci-après sert d'introduction au *Machine Learning*, domaine d'étude de l'Intelligence Artificielle (IA). Cette science, aussi appelée "apprentissage automatique", permet donc à l'ordinateur d'apprendre par lui-même sans qu'il ait été programmé spécifiquement dans ce but au préalable.

L'apprentissage automatique regroupe plusieurs sous-catégories, nommément : l'apprentissage supervisé, l'apprentissage non-supervisé et l'apprentissage de renforcement. Ici, nous ne verrons en application que l'apprentissage supervisé.

Pour résoudre un problème d'apprentissage supervisé, quatre étapes sont à suivre obligatoirement :

Un *Dataset*	Le *dataset* comprend deux types de variables : la variable "objectif", ou *target* notée y ; ainsi qu'une ou plusieurs variables "caractéristiques" notées x.
Un modèle	Un modèle est une représentation mathématique qui va nous permettre de prédire des données dans le cadre d'un problème d'apprentissage automatique. Les modèles peuvent être linéaires où non-linéaires.
Une fonction coût	La fonction coût permet de mesurer la performance de notre modèle en le comparant aux données du *Dataset*.
Un algorithme d'apprentissage	Cette partie va viser à réduire les erreurs de notre modèle qui va permettre de minimiser la fonction coût.

L'une des bibliothèque Python utilisée pour l'apprentissage automatique se nomme Scikit-Learn. Dans un premier temps, nous ne l'utiliserons pas, pour bien comprendre les étapes réalisées. Nous allons donc effectuer, à la suite, une régression linéaire simple, multiple et enfin polynomiale. Par la suite, nous utiliserons Scikit-Learn pour comparer la précision de nos modèles réalisés "à la main".

I. Régression linéaire simple

Consignes :

1) Récupération des données

2) Visualisation des données

3) Création du modèle (model(X, theta))

4) Fonction du coût (fonction_cout(X, Y,theta))

5) Gradient (gradient(X,Y,theta))

6) Descente du gradient (descente_gradient (X, Y, theta, alpha, n_iterations))

7) Evaluer votre modèle en utilisant le coefficient de détermination

8) Tracer la courbe de la fonction du coût selon les itérations

1) Récupération des données

La première étape consiste à importer les bibliothèques nécessaires à l'exploitation de la base de données. On utilise ici pandas, numpy et matplotlib.

Entrée [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Pour la régression linéaire simple, nous utilisons le jeu de données **reg_simple.csv **

Entrée [2]:
```
df = pd.read_csv('Data/reg_simple.csv')
df.head()
```

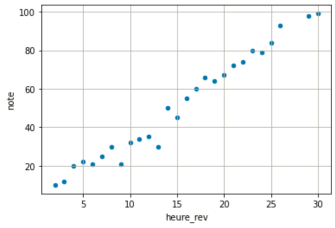
Out[2]:

	heure_rev	note
0	2	10
1	3	12
2	4	20
3	5	22
4	6	21

2) Visualisation des données

Une fois le jeu de données récupéré, il convient de voir si une régression linéaire simple peut être employée à ce dernier. Pour cela, on effectue un affichage graphique en nuages de points :

```
Entrée [9]: dataset.plot.scatter('heure_rev', 'note')
            plt.grid()
```



On observe donc bien une corrélation entre les deux variables.

3) Création du modèle

La visualisation des données nous a permis de confirmer la pertinence de l'usage d'un modèle de régression linéaire.

Notre modèle va donc suivre la forme suivante :

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

Pour intégrer le modèle en langage de programmation, nous allons effectuer une transformation matricielle : pour cela, il faut utiliser numpy. Ainsi, on obtient :

$$f(x) = X.\theta$$

3.1 Création de la matrice X

```
Entrée [228]: #matrice X
              X=np.hstack((x,np.ones(x.shape)))
              X
```

```
Out[228]: array([[ 2.,  1.],
                 [ 3.,  1.],
                 [ 4.,  1.],
                 [ 5.,  1.],
                 [ 6.,  1.],
                 [ 7.,  1.],
                 [ 8.,  1.],
                 [ 9.,  1.],
                 [10.,  1.],
                 [11.,  1.],
                 [12.,  1.],
                 [13.,  1.],
                 [14.,  1.],
                 [15.,  1.],
                 [16.,  1.],
                 [17.,  1.],
                 [18.,  1.],
                 [19.,  1.],
                 [20.,  1.],
```

La matrice X correspond à nos valeurs x à laquelle on ajoute une colonne de biais 1.

3.2 Création du vecteur Thêta

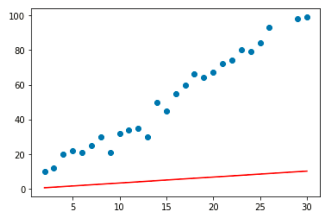
```
Entrée [229]: #initialisation de theta
              theta=np.random.randn(2,1)
              theta.shape
```

Le vecteur Thêta correspond aux valeurs *a* et *b* de notre fonction affine de départ. On prend des paramètres aléatoires pour les définir.

3.3 Visualisation du modèle

Le produit matriciel est effectué par la fonction “.dot”

```
Entrée [230]: def model(X,theta):
                  return X.dot(theta)

Entrée [231]: plt.scatter(x,y)
              plt.plot(x,model(X,theta), color='r')

Out[231]: [<matplotlib.lines.Line2D at 0x16939e101c0>]
```



4) Fonction coût

Comme explicité plus haut, la fonction coût, où erreur quadratique moyenne, va permettre de mesurer les erreurs entre les prédictions f(x) et les valeurs y du jeu de données. Pour éviter de se retrouver avec des valeurs négatives, on calcule le carré de cette différence, où la norme euclidienne :

$$(f(x) - y)^2.$$

La fonction de coût va être la somme de toutes ces différences, soit:

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

On a alors :

```
Entrée [257]: def cost_fonction(X,y,Theta):
                  m=len(y)
                  return 1/(2*m)*np.sum((model(X,theta)-y)**2)

Entrée [233]: cost_fonction(X,y,theta)

 Out[233]: 1359.2385952834486
```

5) Descente de gradient

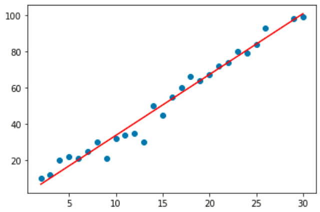
```
Entrée [234]: # gradient
              def grad (X,y,theta):
                  m=len(y)
                  return 1/m * X.T.dot(model(X,theta)-y)

Entrée [235]: # decente de gradient
              def gradient_descent(X,y,theta,learning_rate,n_iterations):
                  for i in range(0,n_iterations):
                      theta=theta-learning_rate* grad(X,y,theta)
                  return theta
```

On va ici utiliser la méthode dite de la descente de gradient pour créer notre algorithme de minimisation. La descente de gradient va nous permettre de calculer les coefficients de a et de b les plus efficaces pour notre modèle.

```
Entrée [236]: # theta final
              theta_final= gradient_descent(X,y,theta,learning_rate=0.001, n_iterations=1000)

Entrée [237]: theta_final

 Out[237]: array([[3.3597416 ],
                  [0.06211433]])
```

Le “Thêta final” correspond aux valeurs d'a et de b les plus optimales pour notre modèle. Pour confirmer ce résultat, on effectue une visualisation graphique du résultat.

```
Entrée [238]: predictions=model(X,theta_final)
              plt.scatter(x,y)
              plt.plot(x,predictions,c='r')
Out[238]: [<matplotlib.lines.Line2D at 0x16939d3b370>]
```



6) Evaluation de notre modèle

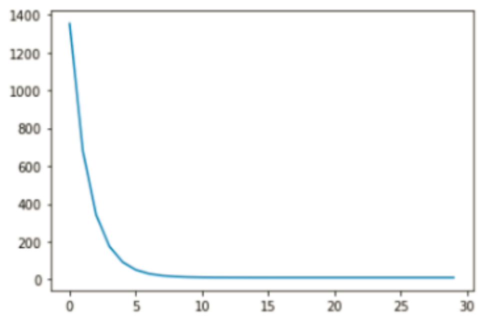
Pour prouver l’efficacité de notre modèle, on calcule de coefficient de détermination :

```
Entrée [239]: def coef_determination(y,pred):
                  u=((y-pred)**2).sum()
                  v=((y-y.mean())**2).sum()
                  return 1-u/v

Entrée [240]: coef_determination(y,predictions)
Out[240]: 0.9732798144975192
```

Le résultat est très proche de 1, notre modèle est donc plutôt précis.

7) Tracer la courbe de la fonction du coût selon les itérations

```
Entrée [17]: plt.plot(range(n_iterations),cost_history)
Out[17]: [<matplotlib.lines.Line2D at 0x2c10f9dfa90>]
```



II. <u>Régression linéaire multiple</u>

<u>Consignes :</u>

1) Implémentez un modèle de régression multiple sur la base de données issue du fichier nommé **boston_house_prices.csv** (sans utiliser la bibliothèque Scikit-learn)

2) Évaluez les résultats obtenus en utilisant la fonction mean_squared_error de sklearn

Pour la régression linéaire multiple, on reprend les mêmes étapes que la régression linéaire simple.

1) Visualisation des données

1- Récupération des données

Entrée [378]:
```
df = pd.read_csv("Data/boston_house_prices.csv")
```

2- Visualisation des données

Entrée [379]:
```
df.head()
```

Out[379]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

2) Transformations matricielles

Entrée [386]:
```
x1=df[['RM']]
x2=df[['LSTAT']]
y=np.array(df[['MEDV']])

y=y.reshape(y.shape[0],1)
y.shape
```

Out[386]: (506, 1)

Entrée [387]:
```
# Faisons la matrice de X
X=np.hstack((x1,x2,np.ones(x1.shape)))
X.shape
```

Out[387]: (506, 3)

3- Création du modèle (model(X,theta))

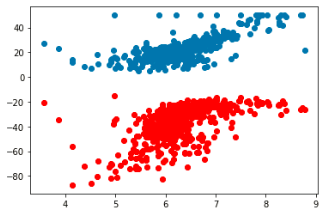
Entrée [388]:
```
#initialisation de theta
theta=np.random.randn(3,1)
theta
```

Out[388]:
```
array([[-1.81459198],
       [-2.12145535],
       [ 0.70719038]])
```

Entrée [389]:
```
def model(x,theta):
    return X.dot(theta)
```

Entrée [390]:
```
plt.scatter(x1,y)
plt.scatter(x1,model(X,theta), color='r')
```

Out[390]: <matplotlib.collections.PathCollection at 0x16939fcf3a0>



4) Calcul de la fonction coût

4- Fonction du coût (fonction_cout(X,Y,theta))

Entrée [391]:
```
#Fonction Cout

def cost_fonction(X,y,theta):
    m=len(y)
    return 1/(2*m)*np.sum((model(X,theta)-y)**2)
```

Entrée [392]:
```
cost_fonction(X,y,theta)
```

Out[392]: 1855.6331265208075

5) Descente de gradient

5- Gradient (gradient(X,Y,theta))

Entrée [393]:
```
def grad (X,y,theta):
    m=len(y)
    return 1/m * X.T.dot(model(X,theta)-y)
```

6- Descente du gradient (descente_gradient(X,Y,theta,alpha,n_iterations))

Entrée [394]:
```
def gradient_descent(X,y,theta,learning_rate,n_iterations):
    cost_history=np.zeros(n_iterations)
    for i in range(0,n_iterations):
        theta=theta-learning_rate* grad(X,y,theta)
        cost_history[i]=cost_fonction(X,y,theta)
    return theta,cost_history
```

6) Evaluation du modèle

7- Evaluer votre modèle en utilisant le coefficient de détermination

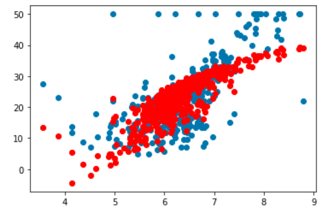
Entrée [395]:
```
n_iterations= 900
learning_rate =0.0007
theta_final,cost_history = gradient_descent(X,y,theta,learning_rate, n_iterations)
```

Entrée [396]:
```
theta_final
```

Out[396]:
```
array([[ 4.67905675],
       [-0.66956482],
       [ 1.60882252]])
```

Entrée [397]:
```python
predictions=model(X,theta_final)
plt.scatter(x1,y)
plt.scatter(x1,predictions,c='r')
```

Out[397]: <matplotlib.collections.PathCollection at 0x16938c729a0>



Entrée [398]:
```python
def coef_determination(y,pred):
    u=((y-pred)**2).sum()
    v=((y-y.mean())**2).sum()
    return 1-u/v
```

Entrée [399]:
```python
coef_determination(y,predictions)
```

Out[399]: 0.6379290745146518

III. <u>Régression linéaire polynomiale</u>

<u>Consignes :</u>

1) En utilisant les bibliothèques adéquates de Python, implémentez un modèle de régression polynomiale sur le jeu de données issu du fichier **Position_Salaire.csv **(sans utiliser la bibliothèque Scikit-learn).

2) Appliquez le même modèle sur le jeu de données issu du fichier data/qualite_vin_rouge.csv

3) Évaluez votre modèle.

*1) Avec le jeu de données **Position salaire.csv***

1.1) <u>Récupération des données</u>

Entrée [400]:
```python
position_salarie=pd.read_csv("Data/Position_Salaries.csv")
print(position_salarie)
```

```
           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
```

Entrée [401]:
```python
# Shape du dataframe
position_salarie.shape
```

Out[401]: (10, 3)

1.2) Transformation matricielle

Entrée [406]:
```
# On définit x et y ; on shape y
x = position_salarie[['Level']]
y = np.array(position_salarie[['Salary']])
y.shape
```
Out[406]: (10, 1)

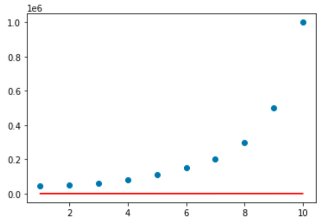
Entrée [407]:
```
# On peut faire la matrice X
X=np.hstack((x**2,x,np.ones(x.shape)))
X.shape
```
Out[407]: (10, 3)

Entrée [408]:
```
# création de  theta
theta=np.random.randn(3,1)
theta
```
Out[408]:
```
array([[ 1.19111933],
       [ 1.10802668],
       [-0.18883726]])
```

Entrée [409]:
```
# création du modèle
def model(x,theta):
    return X.dot(theta)
```

Entrée [410]:
```
# On peut afficher theta
print(theta)
```
```
[[ 1.19111933]
 [ 1.10802668]
 [-0.18883726]]
```

Entrée [411]:
```
# Visualisation
plt.scatter(x,y)
plt.plot(x,model(X,theta), color='r')
```
Out[411]: [<matplotlib.lines.Line2D at 0x16938c87d60>]



1.3) Fonction coût

Entrée [412]:
```
# Création de la fonction coût

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

Entrée [413]:
```
# Appelons la fonction_cout

fonction_cout(X,y,theta)
```
Out[413]: 71432633303.66943

1.4) Descente de gradient

Entrée [414]:
```python
# Création du gradient

def gradient (X,y,theta):
    m=len(y)
    return 1/m * X.T.dot(model(X,theta)-y)
```

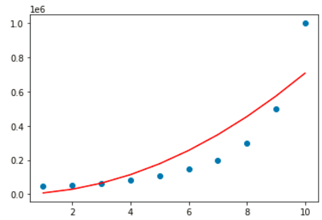
Entrée [415]:
```python
# Création de la fonction descente du gradient

def gradient_descent(X,y,theta,learning_rate,n_iterations):
    cost_history=np.zeros(n_iterations)
    for i in range(0,n_iterations):
        theta=theta-learning_rate* grad(X,y,theta)
        cost_history[i]=cost_fonction(X,y,theta)
    return theta,cost_history
```

Entrée [421]:
```python
# Reprenons theta :
n_iterations= 30
learning_rate =0.0001
theta_final,cost_history = gradient_descent(X,y,theta,learning_rate, n_iterations)

# On affiche theta final
theta_final
```

Out[421]:
```
array([[7025.83295846],
       [ 516.26977162],
       [  34.0756835 ]])
```

Entrée [422]:
```python
predictions=model(X,theta_final)
plt.scatter(x,y)
plt.plot(x,predictions,c='r')
```

Out[422]:
```
[<matplotlib.lines.Line2D at 0x16938ca1400>]
```



Entrée [423]:
```python
def coef_determination(y,pred):
    u=((y-pred)**2).sum()
    v=((y-y.mean())**2).sum()
    return 1-u/v
```

Entrée [424]:
```python
coef_determination(y,predictions)
```

Out[424]:
```
0.8074307706811783
```

*2) Avec le jeu de données **qualite_vin_rouge.csv***

Theta Final

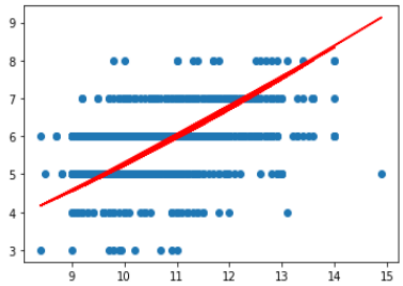
Entrée [14]:
```
n_iterations= 300
learning_rate =0.00001
theta_final,cost_history = gradient_descent(X,y,theta,learning_rate, n_iterations)
```

Entrée [15]:
```
theta_final
```

Out[15]:
```
array([[0.02043089],
       [0.28778999],
       [0.30970857]])
```

Entrée [16]:
```
predictions=model(X,theta_final)
plt.scatter(x,y)
plt.plot(x,predictions,c='r')
```

Out[16]: [<matplotlib.lines.Line2D at 0x1ce781719d0>]



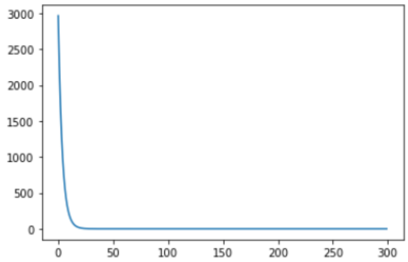
On remarque que le jeu de données n'est pas adapté pour une régression de type Polynomiale.

Courbe d'apprentissage ¶

La regression ne fonctionne pas car les données sont qualitatives et non quantitatives , il serait pertinent de faire une classification ou une regression logistique.

Entrée [20]:
```
plt.plot(range(n_iterations),cost_history)
```

Out[20]: [<matplotlib.lines.Line2D at 0x1ce781d4580>]



Entrée [17]:
```
def coef_determination(y,pred):
    u=((y-pred)**2).sum()
    v=((y-y.mean())**2).sum()
    return 1-u/v
```

Entrée [18]:
```
coef_determination(y,predictions)
```

Out[18]: -0.02810745146807081

Le coefficient est même négatif, le type de régression n'est pas adapté il serait plus judicieux d'appliquer une méthode de classification ou une régression logistique.

IV. Régressions avec Scikit-Learn

Consignes :

1) Refaire les trois régressions avec le module Scikit-Learn

2) Comparer les résultats de précision avec la méthode normale

1) Régression linéaire simple

1 .1) Visualisation des données

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

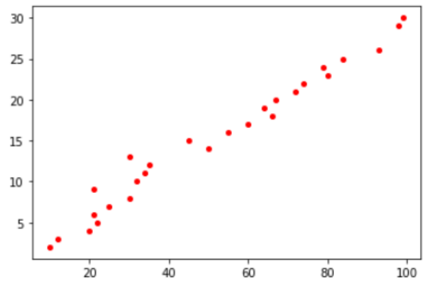
```
df = pd.read_csv('Data/reg_simple.csv')
df.head()
```

	heure_rev	note
0	2	10
1	3	12
2	4	20
3	5	22
4	6	21

```
df.tail()
```

	heure_rev	note
22	24	79
23	25	84
24	26	93
25	29	98
26	30	99

```
plt.plot(df['note'], df['heure_rev'], 'ro', markersize=4)
plt.show()
```



2.2) Transformation matricielle

```
X = df.iloc[:, :-1].values
y = df.iloc[:, 1].values
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
```

```
regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

```
y_pred = regressor.predict(X_test)
```

```
plt.scatter(X_train, y_train, color = 'red')
plt.plot(X_train, regressor.predict(X_train), color = 'blue')
plt.title('Salary vs Experience (Training set)')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()
```

2) Régression linéaire multip

On applique les mêmes étapes que pour la régression linéaire, sauf que notre X contient plus de données.

Une fois les variables X et y définies, on procède à la fragmentation des données en set de train et de test.

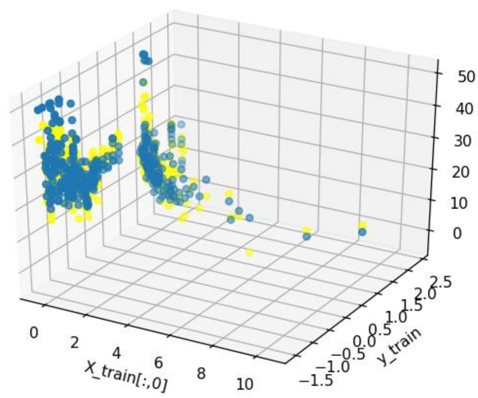
On peut alors standardiser nos valeurs :

```
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
```

L'entraînement des modèles est ensuite identique à celui de la régression précédente.
On peut alors visualiser en 3D nos résultats :

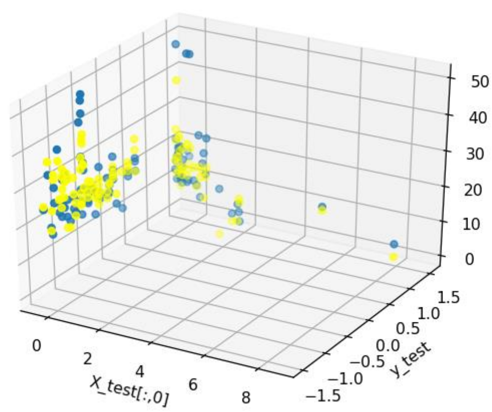
<u>Avec les données train :</u>

```python
from mpl_toolkits.mplot3d import Axes3D
%matplotlib notebook

fig= plt.figure()
ax=fig.add_subplot(111,projection='3d')
plt.xlabel("X_train[:,0]")
plt.ylabel("y_train")
ax.scatter(X_train[:,0],X_train[:,2],y_train)
ax.scatter(X_train[:,0],X_train[:,2],y_pred_train,color = 'yellow')
```



Avec les données test :

```python
from mpl_toolkits.mplot3d import Axes3D
%matplotlib notebook

fig= plt.figure()
ax=fig.add_subplot(111,projection='3d')
plt.xlabel("X_test[:,0]")
plt.ylabel("y_test")
ax.scatter(X_test[:,0],X_test[:,2],y_test)
ax.scatter(X_test[:,0],X_test[:,2],y_pred_test,color = 'yellow')
```



On peut alors évaluer la qualité de notre modèle :

```python
# On utilise RMSE et R²-score.
from sklearn.metrics import r2_score

# Evaluation du modèle (training set)

y_train_predict = regressor.predict(X_train)
rmse = (np.sqrt(mean_squared_error(y_train, y_train_predict)))
r2 = r2_score(y_train, y_train_predict)

print("La performance du modèle pour training set")
print("--------------------------------------")
print('RMSE est de : {}'.format(rmse))
print('R2 score est de : {}'.format(r2))
print("\n")

# Evaluation du modèle (testing set)

y_test_predict = regressor.predict(X_test)
# root mean square error of the model
rmse = (np.sqrt(mean_squared_error(y_test, y_test_predict)))

# Score r-squared pour le modèle
r2 = r2_score(y_test, y_test_predict)

print("La performance du modèle pour testing set")
print("--------------------------------------")
print('RMSE est de : {}'.format(rmse))
print('R2 score est de : {}'.format(r2))
```

Ce qui nous renvoie :

```
La performance du modèle pour training set
--------------------------------------
RMSE est de : 4.396188144698283
R2 score est de : 0.7730135569264233


La performance du modèle pour testing set
--------------------------------------
RMSE est de : 5.783509315085131
R2 score est de : 0.5892223849182514
```

Enfin, la fonction *mean_squared_error* de sklearn nous renvoie le RMSE :

```python
# Root Mean Square Error

from sklearn.metrics import mean_squared_error
import math

MSE = mean_squared_error(y_test, y_test_predict)

RMSE = math.sqrt(MSE)
print("Root Mean Square Error:\n")
print(RMSE)
```

```
Root Mean Square Error:

5.783509315085131
```

3) Régression polynomiale

On importe les bibliothèques et les modules nécessaires :

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
```

On enregistre la data dans un variable appelée « df » dont on peut afficher les premiers éléments :

```
df = pd.read_csv("Data/Position_Salaries.csv")

df.head()
```

	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

On partage les données en X(Level) et y (Salary) :

```
x = df.iloc[:, 1:-1].values
y = df.iloc[:, -1].values
```

Ensuite il faut fractionner nos variables en X_train et X_test et y_train et y_test. On utilise pour cela avec la fonction *train_test_split* de sklearn et l'on sélectionne 80% pour le train et 20% pour le test :

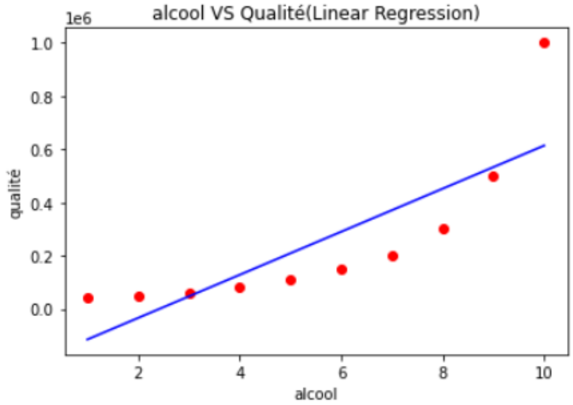
```
#fractionner jeu de données

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)
```

On lance alors le modèle de régression linéaire avec *LinearRegression* :

```
from sklearn.linear_model import LinearRegression
Reg = LinearRegression()
Reg.fit(x, y)
```

Visualisons nos données :

```
plt.scatter(x, y, color = 'red')
plt.plot(x, Reg.predict(x), color = 'blue')
plt.title('alcool VS Qualité(Linear Regression)')
plt.xlabel('alcool')
plt.ylabel('qualité')
plt.show
```



On lance le modèle de régression polynomiale :

```
poly_reg = PolynomialFeatures(degree = 3)
X_poly_train =poly_reg.fit_transform(X_train)
X_poly_test =poly_reg.fit_transform(X_test)
```

```
lin_reg_2 = LinearRegression()
lin_reg_2.fit(X_poly_train, y_train)
```

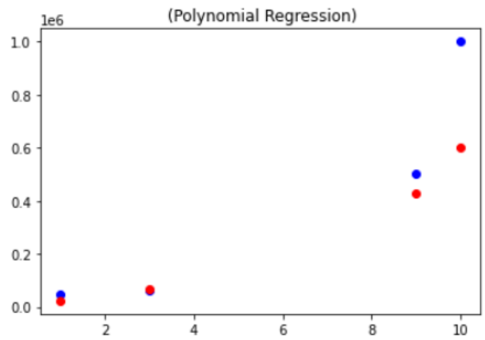
```
LinearRegression()
```

On procède à la phase de prédiction :

```
y_pred = lin_reg_2.predict(X_poly_test)
```

En visualisant notre jeu de données, nous obtenons le graphique suivant (du fait de la faible quantité du jeu de données) :

```
plt.scatter(X_test, y_test, color = 'blue')
plt.scatter(X_test, y_pred, color = 'red')
plt.title(' (Polynomial Regression)')
plt.xlabel('')
plt.ylabel('')
plt.show()
```



Evaluons notre modèle avec le coefficient de détermination avec *mean squarred error* avec la fonction *r2_score* de *sklearn.metrics* :

```
from sklearn.metrics import r2_score
r2 = r2_score(y_test, y_pred)
print('R2 score est de : {}'.format(r2))
```

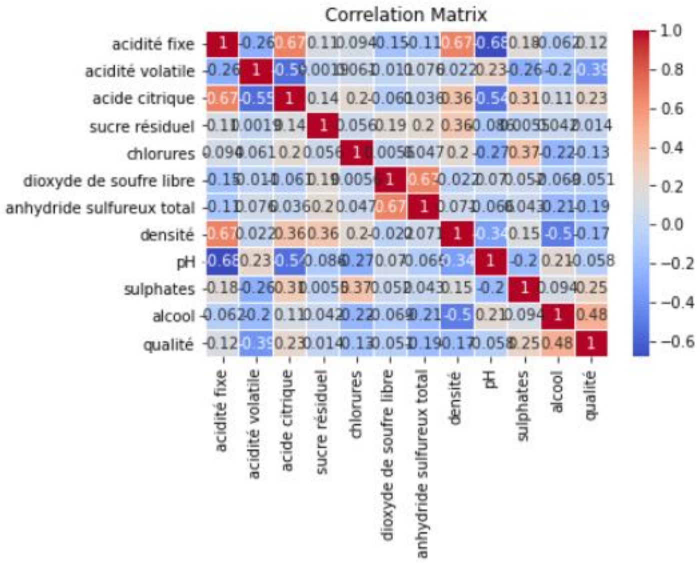
```
R2 score est de : 0.8688814279352782
```

4) Régression polynomiale avec les données « qualité_vin_rouge.csv »

Cette régression est dans l’ensemble identique à la précédente, aussi nous ne préciserons que les légères variations.

```
# matrice de corrélation

plt.title("Correlation Matrix")
sns.heatmap(data=df.corr(),annot=True,cmap='coolwarm',linewidths=0.1)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b34ca4e6d0>
```

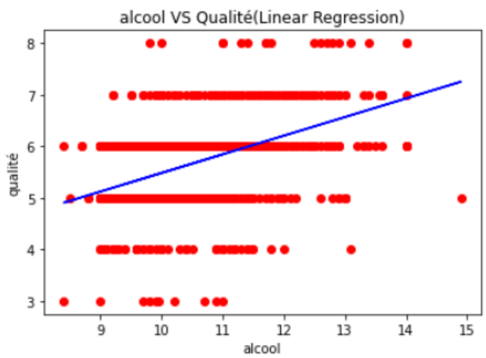


En voyant cette image, on peut sélectionner uniquement la variable "alcool" (0,48).

En faisant une matrice de corrélation nous pouvons constater que la variable la plus intéressante est « alcool », corrélée à 0,48. C’est donc cette variable que nous mettrons dans notre X.

```
#On peut visualiser les données

plt.scatter(x, y, color = 'red')
plt.plot(x, LinReg.predict(x), color = 'blue')
plt.title('alcool VS Qualité(Linear Regression)')
plt.xlabel('alcool')
plt.ylabel('qualité')
plt.show
```

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



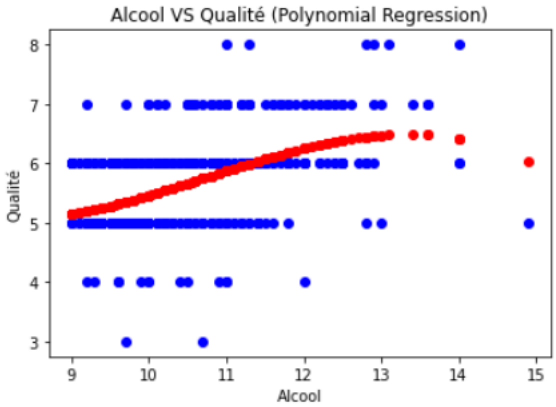
```python
# On lance le modèle de régression polynomiale

poly_reg = PolynomialFeatures(degree = 3)
X_poly_train =poly_reg.fit_transform(X_train)
X_poly_test =poly_reg.fit_transform(X_test)
```

```python
lin_reg_2 = LinearRegression()
lin_reg_2.fit(X_poly_train, y_train)
```

```
LinearRegression()
```

```python
# Phase de prédiction
y_pred = lin_reg_2.predict(X_poly_test)
```

```python
plt.scatter(X_test, y_test, color = 'blue')
plt.scatter(X_test, y_pred, color = 'red')
plt.title('Alcool VS Qualité (Polynomial Regression)')
plt.xlabel('Alcool')
plt.ylabel('Qualité')
plt.show()
```



Nous pouvons évaluer notre modèle avec avec le coefficient de détermination avec *mean squarred error* avec la fonction *r2_score* de *sklearn.metrics* :

```python
# Evaluation du modèle avec coef de détermination avec mean squarred error

from sklearn.metrics import r2_score
r2 = r2_score(y_test, y_pred)
print('R2 score est de : {}'.format(r2))
```

```
R2 score est de : 0.18766066772337553
```

Conclusion :

Premier projet d’apprentissage automatique, plutôt ardu. J’ai encore beaucoup de choses à assimiler, mais cela m’a permis de démystifier le principe.