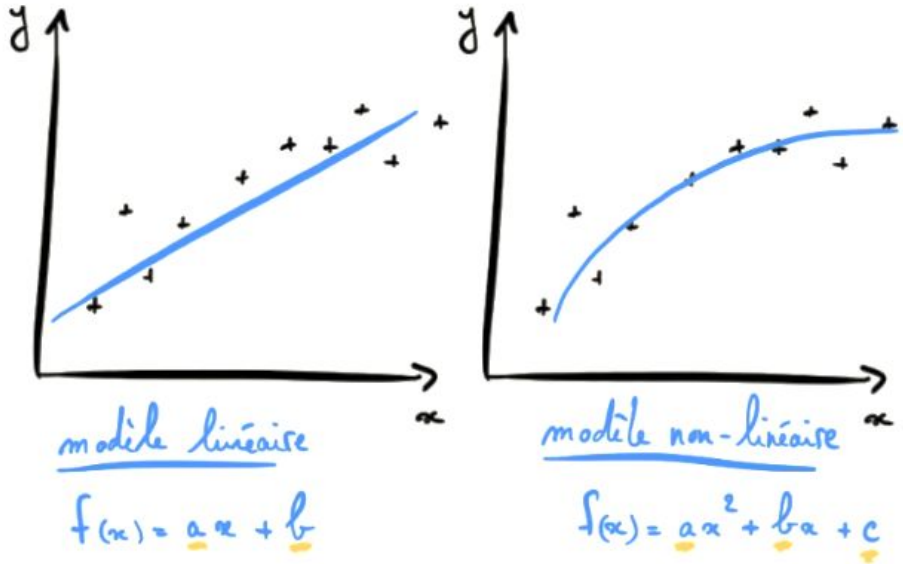
Premier Model IA

Aude Patricia

Sommaire

- Un Rappel sur la régression linéaire simple multiple et polynomiale
- Explication de chaque fonction en présentant l'équation matricielle qui lui correspond (par la méthode normale)
- Présentation des résultats des modèles sous forme de graphiques
- Evaluation des modèles
- Présentation des résultats avec le module Scikit Learn
- Comparaison avec la méthode normale
- Conclusion Qu'avez vous appris? Comment? Des difficultés? Comment vous sentez après ce projet)

Les modèles



On définit a, b, c, etc. comme étant les **paramètres** d'un modèle.

Régression Linéaire

Cette régression cherche à établir, sous forme d'une droite, u**ne relation entre une variable expliquée et une variable explicative.**

Par exemple dans l'exemple que nous verrons plus tard, prédire une note à un examen (variable expliquée) en fonction du nombre d'heures de révisions (variable explicative).

En d'autres termes, les données d'une série d'observations sont représentées sous forme d'un nuage de points et l'on cherche à trouver une droite passant au plus près de ces points.

La régression linéaire

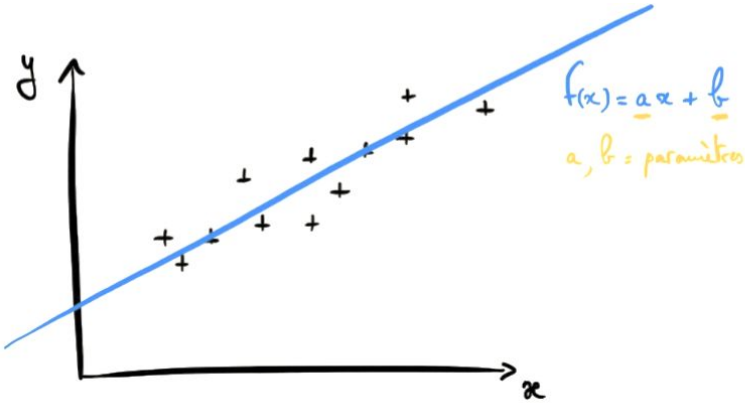
$$h_\theta(x) = \theta_0 + \theta_1 x$$

Développer un programme de régression linéaire

- récolter les données X,y $avec X,y \in \mathbb{R}^{m\times 1}$
- créer un modèle linéaire $F(X)=X.\theta o\grave{u}\theta=(ab)$
- définir la fonction coût
- trouver des paramètres qui minimisent la fonction coût

Étapes pour programmer les régressions :

- importer les librairies
- créer un dataset
- développer et entraîner le modèle



La régression linéaire

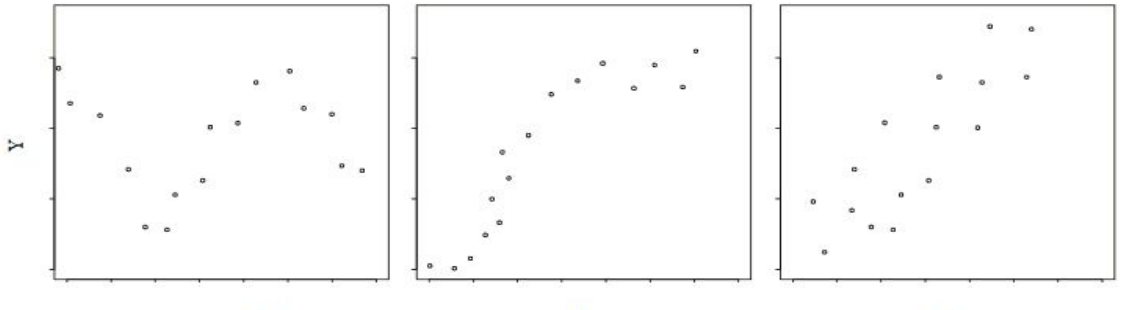
But :

expliquer une variable Y à partir d'une variable X

chercher une fonction f telle que $y_i \approx f(x_i)$.

établir un tracé des observations pour savoir si le modèle est pertinent (droite qui passe au plus près des nuages de points pour obtenir un coefficient directeur)

Graphique 1 :



La régression multiple

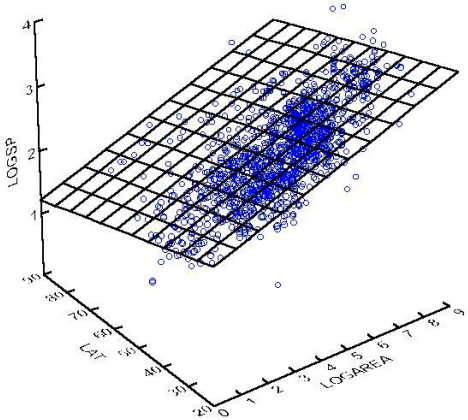
Ici **nous allons utiliser plusieurs variables explicatives** (contrairement à la linéaire).

Une étape importante lors de l'utilisation de multiples variables explicatives est leur normalisation (mise à l'échelle (scaling). Cette étape va consister à faire en sorte que la moyenne de chaque série d'observations soit égale à 0, que la variance et l'écart type soient égaux à 1. Cette méthode est également appelée centrage de réduction .

Une fois cette étape réalisée, nous pouvons passer à la prédiction grâce à la méthode de descente de gradient ou bien encore la méthode des moindes carrés. Ces deux méthodes prenant en compte les différentes variables explicatives mises à l'échelle dans le but de prédire la variable expliquée.

La régression multiple

- outil mis en oeuvre pour l'étude des données multidimensionnelles
- Multiples paramètres (taille, nombre de chambres, nombre d'étages...)



$$Y = \beta_0 + \beta_1 X_1 + \ldots + \beta_k X_k + \varepsilon$$

où β_j = paramètres fixes (mais inconnus)

ε = terme aléatoire : de moyenne 0

d'écart - type σ

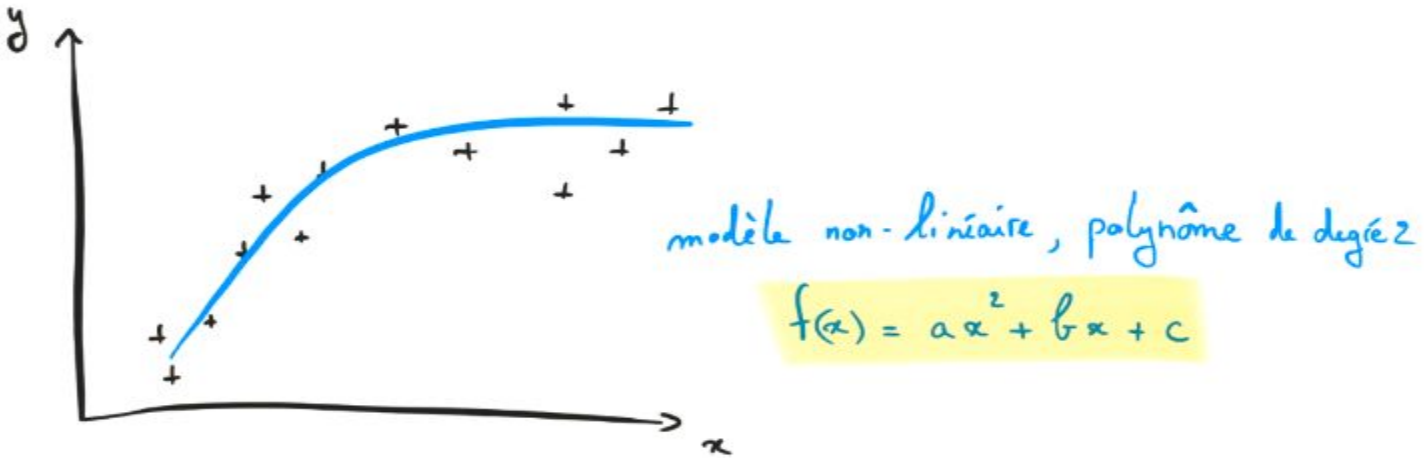
Y	$X_1 X_2 \ldots X_k$
Variable à expliquer	Variables explicatives
Variable dépendante	Variables indépendantes
Variable endogène	Variables exogènes

La régression polynomiale

Il est parfois difficile de trouver une droite pouvant passer parmi les points de la série d'observations de façon optimale. Cependant, il est **parfois possible de trouver un lien entre les variables à l'aide d'une courbe**. C'est ce que permet la régression polynomiale en ajoutant des "plis" à la courbe à l'aide d'éléments appelés pôlynomes.

La régression polynomiale

- approche statistique employée pour modéliser une forme non linéaire entre X (variable explicative) et Y (réponse)
- évaluer la linéarité et prédictions



Explication de chaque fonction en présentant l'équation matricielle qui lui correspond (par la méthode normale)

```
def model(X, theta):
    return X.dot(theta) # la fonction nous retourne le produit matriciel de X par
theta
 # fonction coût
def cost_function(X, y, theta): # on calcule la fonction coût qui est l'erreur
quadratique moyenne
    m = len(y) # nombre d'exemple qu'on a dans notre datasale et qui est aussi long
que le vecteur y
    return 1/(2*m) * np.sum((model(X, theta) - y)**2) # carré de la différence entre
notre modele et y
print(cost_function(X, y, theta))
# fonction calcul du gradient
def grad(X, y, theta):
    m = len(y)
    return 1/m * X.T.dot(model(X, theta) - y)#XT transposé de x
# descente de gradient
def gradient_descent(X, y, theta, learning_rate, n_iterations): #learning rate
variable alpha,
```

```
    # création d'un tableau de stockage pour enregistrer l'évolution du Cout du modele
    cost_history = np.zeros(n_iterations) # ceci permet  de créer un tableau rempli de 0 et il est aussi long que nos itérations

    for i in range(0, n_iterations):
        theta = theta - learning_rate * grad(X, y, theta) # mise a jour du parametre theta (formule du gradient descent) pendant n itérations
        cost_history[i] = cost_function(X, y, theta) # on enregistre la valeur du Cout au tour i dans cost_history[i]

    return theta, cost_history
```

Formules

$$h_\theta(x) = \theta_0 + \theta_1 x$$

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$$

Présentation des résultats avec le module Scikit Learn

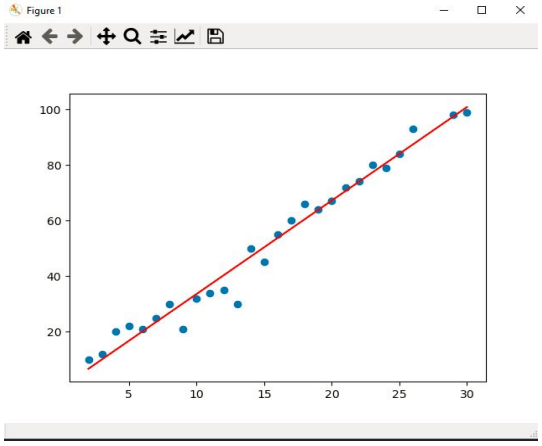
trame exploitations des données via scikit learn

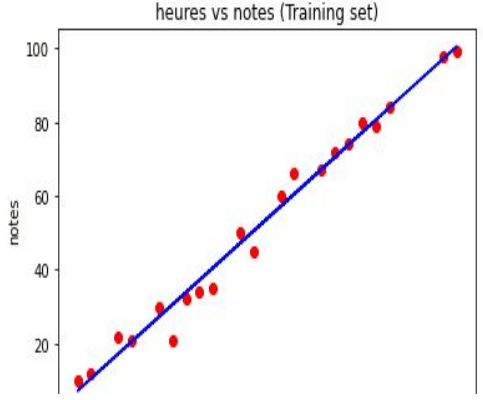
```
model = LinearRegression()

model.fit (X,y)

model.score(X, y)

model.predict (X)
```

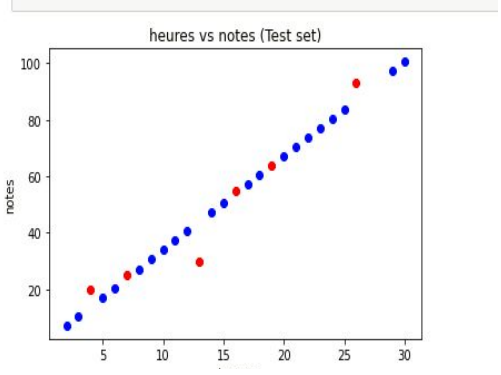
1) Sélectionner un estimateur et précision de ses hyperparamètres.
2) Entrainer le modèle sur les données (X,y)
   X et Y doivent avoir 2 dimensions (n samples, n features)
3) Evaluer le model
4) Utiliser le model

Présentation des résultats des modèles sous forme de graphiques

linéaire





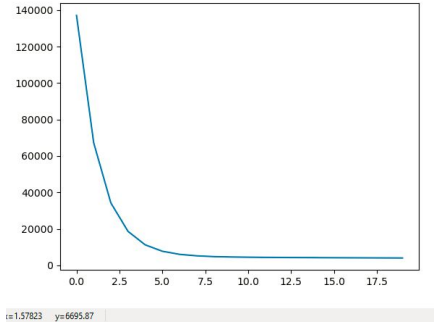



```
(27, 2)
theta : [[-0.20242424]
 [ 0.46684298]]
1854.0894329623372
theta_final : [[3.32628691]
 [0.64777232]]
R2 :  1.0000000501458892
```

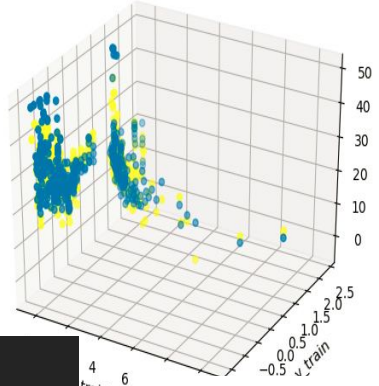
```
Entrée [30]: # Evaluation du modèle par le mean_squared_error et mean_absolute_error
             mse = mean_squared_error(y_test, y_pred)
             mae = mean_absolute_error(y_test, y_pred)

             print(mse)
             print(mae)

             44.37111207555747
             4.705161854768154

Entrée [ ]: plus le mse est proche de 0 mieux c'est

Entrée [31]: regressor.score(X_test, y_test)

  Out[31]: 0.9328868520347855
```

Présentation des résultats des modèles sous forme de graphiques

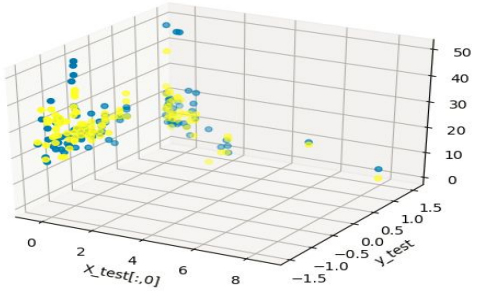
Régression Linéaire Multiple



train



test

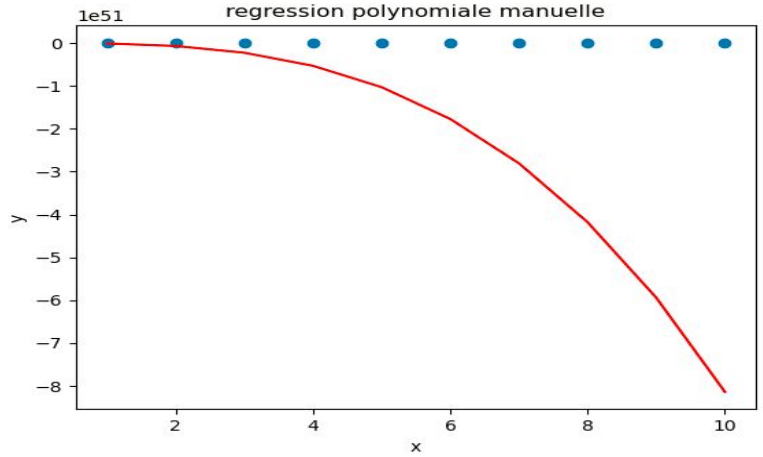


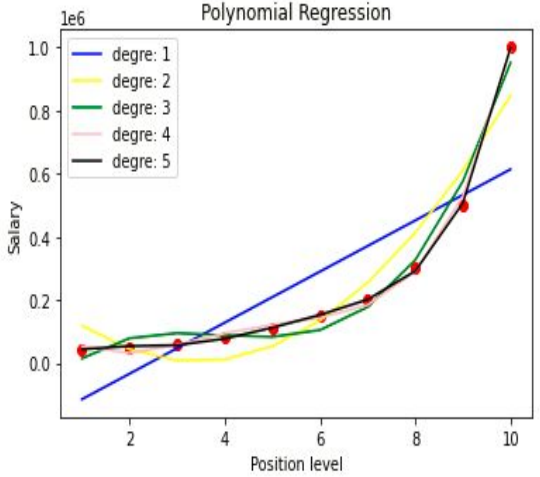

```
      CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD  TAX  PTRATIO       B  LSTAT  MEDV
0  0.00632  18.0   2.31     0  0.538  6.575  65.2  4.0900    1  296     15.3  396.90   4.98  24.0
1  0.02731   0.0   7.07     0  0.469  6.421  78.9  4.9671    2  242     17.8  396.90   9.14  21.6
2  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
['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
(506, 13)
(506, 1)
(506, 14)
mean squared error :8071.631877303024
```

```
Out[10]: <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x23d2fc0f220>

Entrée [11]:
#Evalution du modèle par le mean_squared_error

mse2 = mean_squared_error(y_test, y_pred_test)
print(mse2)

33.448979997767648

Entrée [36]: regressor.score(X, y)

Out[36]: 0.9733112140083816
```

Présentation des résultats des modèles sous forme de graphiques

Regression polynomiale





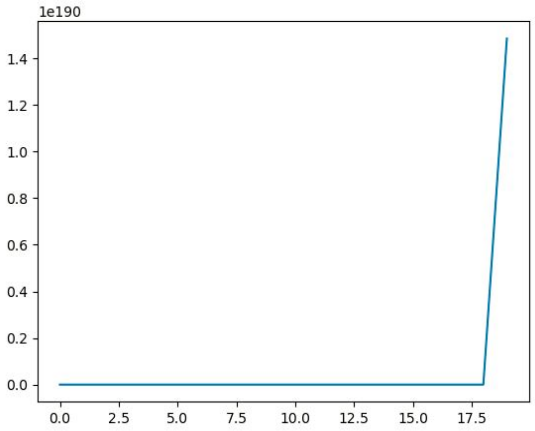

```
ojets/projet_5_ia/multi.py"
(10, 1)
(10, 4)
mean squared error :1.3087989271533603e+103
```

```
Entrée [15]: lin_reg_2.score(X_poly, y)

  Out[15]: 0.9997969027099755
```

```
Entrée [16]: mse2 = mean_squared_error(y, y_pred)
             mse2

  Out[16]: 26695878787.878788
```

Présentation des résultats des modèles sous forme de graphiques

Regression polynomiale VIN : données difficilement exploitables




```
Entrée [27]: #Fractionnement le jeu de données en jeu d'entraînement et jeu de test (20% pour le test)
             from sklearn.model_selection import train_test_split
             X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1/5, random_state = 0)

Entrée [28]: #modèle LinearRegression et entraînez le sur les données d'entraînement
             from sklearn.linear_model import LinearRegression
             regressor = LinearRegression()
             regressor.fit(X_train, y_train)

  Out[28]: LinearRegression()

Entrée [36]: # Predicting the Test set results
             y_pred = regressor.predict(X_test)

Entrée [37]: regressor.score(X,y)

  Out[37]: 0.3593724283394686

Entrée [41]: mse2 = mean_squared_error(y_test,y_pred)
             print(mse2)

             0.38447119782012484
```

```
PS C:\Users\utilisateur\Google Drive\microsoft_ia\Google
PS C:\Users\utilisateur\Google Drive\microsoft_ia\Google
ojets/projet_5_ia/multi 2.py"
(1599, 11)
(1599, 1)
(1599, 22)
(1599, 23)
```

Conclusion

A travers ce projet, nous avons pu :

- comprendre la différence entre les 3 régressions
- vérifier la nécessité d'avoir un jeu de données limpide
- découvrir la façon dont le mse est calculé avant d'utiliser scikit learn
- la facilité d'utilisation de scikit learn
-