Régression Linéaire

1 Introduction

Ce tutoriel fait suite au document consacré aux méthodes d'approximation d'un modèle de régression linéaire. Nous travaillons sous Python avec le package « scikit-learn ». En effet, nous effectuons une comparaison des méthodes d'approximation en commençant par la régression linéaire multiple usuelle telle qu'elle est proposée dans la librairie « **Scikit-learn** ». Ains, nous allons aborder dans un premier temps la régression linéaire, ensuite nous utiliserons les techniques de régularisation[1] telles que la régression Ridge, Lasso et Elastic net. Toutes ces méthodes seront utilisées dans l'unique but de constater l'amélioration du modèle utilisé, afin de pouvoir faire de prédiction sur des données futures que le modèle n'a jamais vues auparavant. Donc, avant d'utiliser notre modèle d'une manière définitive sur des données futures, il faut nous assurer de ses performances. L'ensemble de données de nos jours est de plus grand et complexe, et surtout nous voudrions y tirer le plus d'informations possibles…

Ce tutoriel est organisé de la manière suivante :

- La régression linéaire
- La régression Ridge
- La régression Lasso
- La régression Elastic net

Avant d'entrer dans le vif du sujet, nous allons explorer la base de données qui sera utilisée dans ce tutoriel.

2 Données

Nous avons récupéré l'ensemble de données sur les prix des logements à Boston de la librairie **Scikit-learn**. Nous pouvons charger cet ensemble de données en invoquant la librairie load_boston. L'**objectif** de ce tutoriel est d'observer les **performances** de **modèles** utilisés afin d'**expliquer** les **prix des logements** à **Boston** à partir de **13 caractéristiques** (variables). Cet ensemble de données comprend **506 observations**. Pour avoir plus des détails sur l'ensemble de données veillez clique ici.

Avant de charger l'ensemble de données, il faut tout d'abord importer les librairies suivantes :

```
# Importation des bibliothèques nécessaires
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import dataset
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, RidgeCV, Ridge, Lasso, Elastic-
Net, ElasticNetCV
```

Sous Python, nous chargeons notre ensemble de données dans **Scikit-learn** et nous le transformons en Dataframe comme suit :

```
#importer l'ensemble de données
df = datasets.load_boston()
```

[1] Referez vous à notre dernière publication pour avoir plus de détails sur les techniques de régularisation.

```
# Description data
print(df.DESCR)
# Transformation des données sous forme Dataframe pandas
df_data = pd.DataFrame(df.data,columns=df.feature_names)
#Ajout de la colonne cible (target)
df_data['House Price'] = pd.Series(df.target)
# Afficher les 5 premières lignes du datatset
df_data.head()
```

Dans le tableau ci-dessous nous affichons les cinq premières lignes de notre jeu de données :

Tableau 1 : Les cinq premières lignes de notre dataset

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	House Price
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

Avant de commencer à implémenter les algorithmes de régression, nous allons tout d'abord nous assurer que nos données ne contiennent pas des valeurs manquantes, sont mises à l'échelle et nous nous assurons également de la corrélation que puisse exister entre nos variables.

La première ligne de code ci-dessous nous indique, dans le tableau ci-après, qu'il n'y a aucune valeur manquante dans nos données.

```
# Repérer les valeurs manquantes
print(df_data.isna().sum())
```

Tableau 2 : Valeurs manquantes

```
CRIM          0
ZN            0
INDUS         0
CHAS          0
NOX           0
RM            0
AGE           0
DIS           0
RAD           0
TAX           0
PTRATIO       0
B             0
LSTAT         0
House Price   0
```

Comme nous l'avons déjà dit, nous constatons dans le tableau ci-dessous qu'il n'y a aucune valeur manquante.

Avec la ligne de code ci-dessous, nous affichons un tableau de quelques statistiques descriptives de notre jeu de données.

```
#Statistques descriptives
print(df_data.describe())
```

Tableau 3 : Statistiques descriptives

```
              CRIM          ZN       INDUS  ...           B       LSTAT  House Price
count  506.000000  506.000000  506.000000  ...  506.000000  506.000000   506.000000
mean     3.613524   11.363636   11.136779  ...  356.674032   12.653063    22.532806
std      8.601545   23.322453    6.860353  ...   91.294864    7.141062     9.197104
min      0.006320    0.000000    0.460000  ...    0.320000    1.730000     5.000000
25%      0.082045    0.000000    5.190000  ...  375.377500    6.950000    17.025000
50%      0.256510    0.000000    9.690000  ...  391.440000   11.360000    21.200000
75%      3.677083   12.500000   18.100000  ...  396.225000   16.955000    25.000000
max     88.976200  100.000000   27.740000  ...  396.900000   37.970000    50.000000

[8 rows x 14 columns]
```

Dans le tableau ci-dessous, nous avons les valeurs moyennes, minimales, maximales … de toutes nos variables.

A présent, nous allons calculer et afficher la corrélation entre nos variables via les lignes de codes ci-dessous :

```
# Calculer et afficher la correlation entre les differentes variables
corr = df_data.corr()
corr.style.background_gradient(cmap='coolwarm').set_precision(2)
```

Figure 1 : Visualisation de la corrélation

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	House Price
CRIM	1.00	-0.20	0.41	-0.06	0.42	-0.22	0.35	-0.38	0.63	0.58	0.29	-0.39	0.46	-0.39
ZN	-0.20	1.00	-0.53	-0.04	-0.52	0.31	-0.57	0.66	-0.31	-0.31	-0.39	0.18	-0.41	0.36
INDUS	0.41	-0.53	1.00	0.06	0.76	-0.39	0.64	-0.71	0.60	0.72	0.38	-0.36	0.60	-0.48
CHAS	-0.06	-0.04	0.06	1.00	0.09	0.09	0.09	-0.10	-0.01	-0.04	-0.12	0.05	-0.05	0.18
NOX	0.42	-0.52	0.76	0.09	1.00	-0.30	0.73	-0.77	0.61	0.67	0.19	-0.38	0.59	-0.43
RM	-0.22	0.31	-0.39	0.09	-0.30	1.00	-0.24	0.21	-0.21	-0.29	-0.36	0.13	-0.61	0.70
AGE	0.35	-0.57	0.64	0.09	0.73	-0.24	1.00	-0.75	0.46	0.51	0.26	-0.27	0.60	-0.38
DIS	-0.38	0.66	-0.71	-0.10	-0.77	0.21	-0.75	1.00	-0.49	-0.53	-0.23	0.29	-0.50	0.25
RAD	0.63	-0.31	0.60	-0.01	0.61	-0.21	0.46	-0.49	1.00	0.91	0.46	-0.44	0.49	-0.38
TAX	0.58	-0.31	0.72	-0.04	0.67	-0.29	0.51	-0.53	0.91	1.00	0.46	-0.44	0.54	-0.47
PTRATIO	0.29	-0.39	0.38	-0.12	0.19	-0.36	0.26	-0.23	0.46	0.46	1.00	-0.18	0.37	-0.51
B	-0.39	0.18	-0.36	0.05	-0.38	0.13	-0.27	0.29	-0.44	-0.44	-0.18	1.00	-0.37	0.33
LSTAT	0.46	-0.41	0.60	-0.05	0.59	-0.61	0.60	-0.50	0.49	0.54	0.37	-0.37	1.00	-0.74
House Price	-0.39	0.36	-0.48	0.18	-0.43	0.70	-0.38	0.25	-0.38	-0.47	-0.51	0.33	-0.74	1.00

Dans le tableau ci-dessous, nous constatons que la corrélation entre certaines de nos variables est très élevée positivement que négativement. Par exemple, les variables **INDUS** et **NOX** sont très corrélées positivement (**76%**), les variables **NOX** et **DIS** sont très corrélées négativement (**77%**). C'est là qu'intervienne la méthode de **régularisation** afin de réduire ou éliminer les coefficients de variables qui sont corrélées. En effet, **Ridge** réduit les valeurs de coefficients et effectue une sélection groupée de variables corrélées, tandis que le **Lasso** élimine carrément les coefficients des variables corrélées, on dit que le Lasso effectue une **sélection de caractéristiques** et finalement l'**Elastic Net** combine les deux techniques de Ridge et Lasso, pour pallier surtout aux limites de Lasso, quand il s'agit de sélection de variables corrélées.

Ensuite, nous affichons ci-dessous la **corrélation** de notre **variable cible** avec les **variables indépendantes**. Il est très important de connaitre les variables qui ont une forte liaison avec la variable cible. Nous affichons les valeurs de cette corrélation d'une manière décroissante pour constater rapidement quelles sont les variables le plus corrélées avec notre variable cible.

```
# Verifier le lien entre la variable cible et les variables independantes
corr_target = df_data.corr()['House Price']
print((corr_target).sort_values())
```

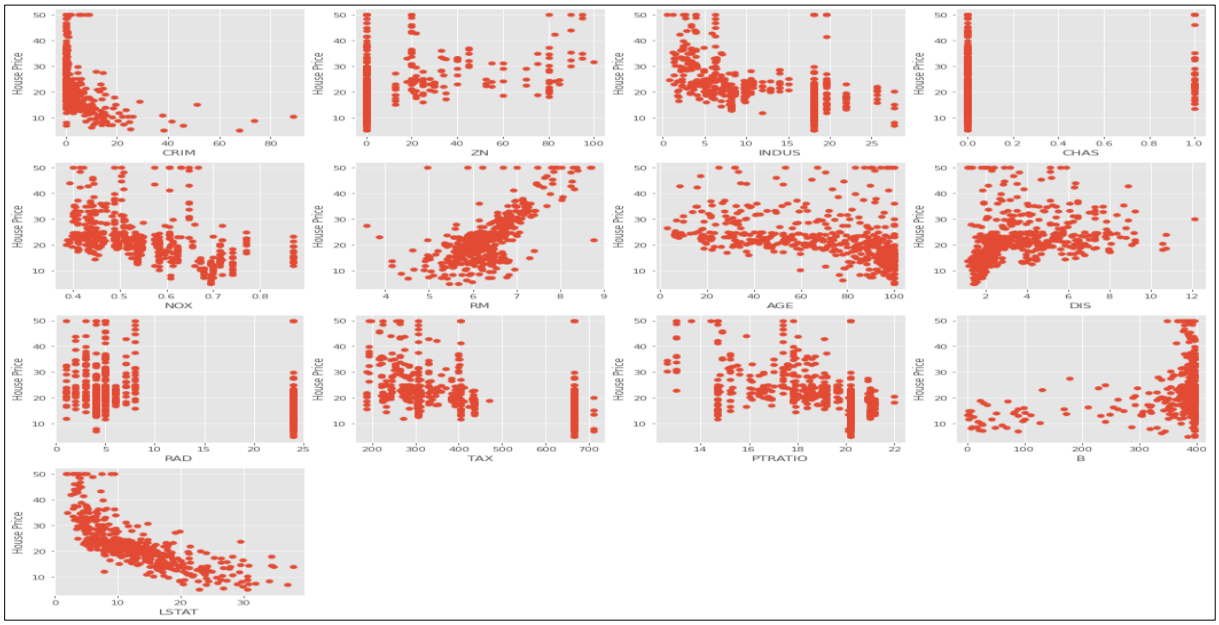
Tableau 4 : Corrélation entre la variable cible et les différentes variables indépendantes

```
LSTAT          -0.737663
PTRATIO        -0.507787
INDUS          -0.483725
TAX            -0.468536
NOX            -0.427321
CRIM           -0.388305
RAD            -0.381626
AGE            -0.376955
CHAS            0.175260
DIS             0.249929
B               0.333461
ZN              0.360445
RM              0.695360
House Price     1.000000
Name: House Price, dtype: float64
```

Nous constatons que les variables le plus corrélées avec notre variable cible (**House price**) sont dans l'ordre : **RM**, **ZN**, **B**, **DIS**, etc. On peut alors dire que ces variables particulières seront plus significatives dans l'analyse de prix de logements que les autres variables indépendantes. Nous pouvons visualiser cette corrélation pour mieux observer le lien entre notre variable cible et les variables explicatives.

```
# Generer un nuage de poiunts entre les variables indépendantes et dépendantes
plt.style.use('ggplot')
fig = plt.figure(figsize = (18, 18)
for index, feature_name in enumerate(df.feature_names):
    ax = fig.add_subplot(4, 4, index + 1)
    ax.scatter(df.data[:, index], df.target)
    ax.set_ylabel('House Price', size = 12)
    ax.set_xlabel(feature_name, size = 12)
plt.show()
```

Figure 2 : Visualisation de la corrélation entre la variable target et les différentes caractéristiques



Nous pouvons observer à partir des diagrammes de dispersion ci-dessus que certaines des variables indépendantes ne sont pas très corrélées (positivement ou négativement) avec la variable cible. Ces variables verront leurs coefficients diminuer en régularisation. De plus, ce diagramme ne fait que confirmer le tableau de corrélation que nous avons réalisé entre les variables indépendantes et la variable cible ci-dessous.

Maintenant que nous avons analysé l'ensemble de données, nous en savons suffisamment sur nos données pour choisir des modèles qui conviendront mieux à la compréhension des modèles.

Dans ce cas, nous essayons de prédire une valeur de variable continue basée sur les variables indépendantes, c'est donc un problème de régression. Pour ce faire, nous utiliserons certaines techniques expliquées dans ma dernière publication, mais d'abord, nous devons diviser les données en un ensemble d'entrainement et de test[2] : nous divisons nos données avec 70% des données d'entrainement et 30% des données de test. Pour avoir plus d'informations sur la répartition de données, veuillez cliquer ici et ici.

```
# Mettons toutes les variables dependantes dans X et la variable cible dans y
X = df_data.iloc[:, :-1]
y = df_data.iloc[:, -1]
```

[2] Il est impératif de diviser les données en un ensemble d'entrainement et de test dans un domaine d'apprentissage automatique. Ainsi, nous utilisons le modèle avec les données d'entrainement et nous tester les performances du modèle sur les données du test.

```
# appliquons la répartition sur nos données X et y
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42)
print("La forme de notre ensemble d'entrainement X = % s et y = % s : "%(
    X_train.shape, y_train.shape))
print("La forme de notre ensemble de test X = % s et y = % s : "%(
    X_test.shape, y_test.shape))
```

C'est maintenant le bon moment pour tester les modèles. Nous utiliserons d'abord la **régression linéaire multiple**. Nous entraînons le modèle sur les données d'entraînement et calculons le $\mathbf{R^2}$, le **MSE** sur les données de test.

3 La régression linéaire multiple

Voici ci-dessous les lignes de codes pour la régression linéaire multiple :

```
# Instancier le modèle de régression
reg = LinearRegression()
#Ajuster le modele
reg.fit(X_train, y_train)
# Gener de prediction sur l'ensemble du test
reg_pred = reg.predict(X_test)
# Calculer l'erreur quadratique moyenne (mse)
mean_squared_error = np.mean((reg_pred - y_test)**2)
print("L'erreur quadratique moyenne : ", mean_squared_error)
print("Coefficient de determination: %.2f" % r2_score(y_test, reg_pred))
# Rassembler les coefficients et leurs noms de variables correspondants dans un Da-
taframe
reg_coefficient = pd.DataFrame()
reg_coefficient["Columns"] = X_train.columns
reg_coefficient['Coefficient Estimate'] = pd.Series(reg.coef_)
#Afficher les coefficients
print((reg_coefficient))
```

La sortie de ces lignes de codes ci-dessus donne le tableau ci-après :

Tableau 5 : Coefficients du modèle de régression multiple et les metrics d'évaluation

```
L'erreur quadratique moyenne :  21.51744423117765
Coefficient de determination: 0.71
    Columns  Coefficient Estimate
0      CRIM            -0.133470
1        ZN             0.035809
2     INDUS             0.049523
3      CHAS             3.119835
4       NOX           -15.417061
5        RM             4.057199
6       AGE            -0.010821
7       DIS            -1.385998
8       RAD             0.242727
9       TAX            -0.008702
10  PTRATIO            -0.910685
11        B             0.011794
12    LSTAT            -0.547113
```

Nous avons ainsi une **erreur quadratique moyenne** de 21,52, et un coefficient de détermination de 71%. Ce qui veut dire que le modèle explique près de 71% de la variabilité de la variable cible (House price). En revanche, **Scikit-learn** ne calcule pas directement les p-values pour constater la significativité de coefficients, mais ici notre objectif c'est d'abord les performances de notre modèle. Donc, nous allons

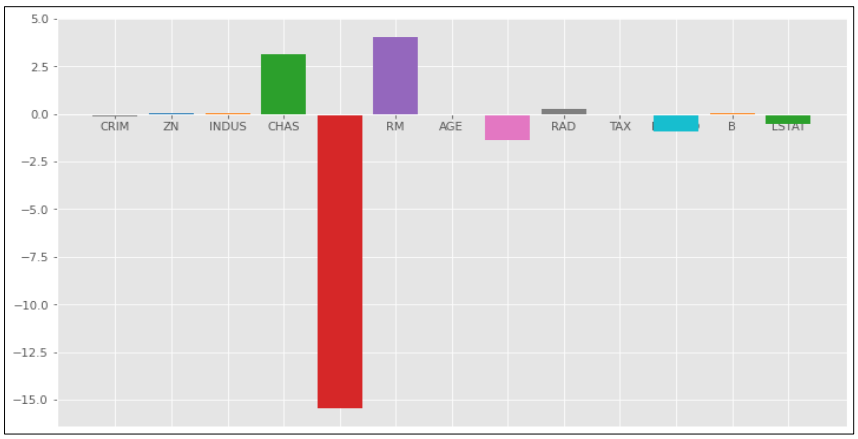
nous contenter des metrics d'évaluation (**mse** et $\mathbf{R^2}$), ces dernières nous permettront d'évaluer les performances de notre modèle. Pour les cieux qui veulent plus de détails sur le résultat de la régression linéaire (OLS), veuillez consulter le module statsmodels.

Nous allons à présent tracer un graphique à barres des coefficients ci-dessus à l'aide de la bibliothèque de visualisation matplotlib.

```
# Définir la figure et l'axe
fig, ax = plt.subplots(figsize =(12, 7))
#Instancier les couleurs
color =['tab:gray', 'tab:blue', 'tab:orange',
'tab:green', 'tab:red', 'tab:purple', 'tab:brown',
'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan',
'tab:orange', 'tab:green', 'tab:blue', 'tab:olive']
#Tracer les coefficients en diagramme en barres
ax.bar(reg_coefficient["Columns"], reg_coefficient['Coefficient Estimate'], co-
lor = color)
ax.spines['bottom'].set_position('zero')
plt.style.use('ggplot')
plt.show()
```

Nous affichons les variables indépendantes avec leurs coefficients respectifs ci-dessous :

Figure 3 : Visualisation des coefficients de la régression linéaire

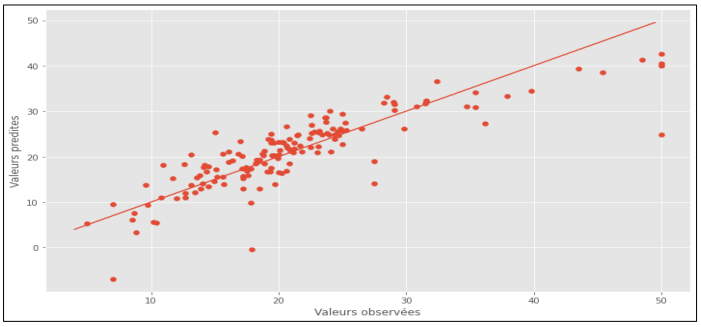


Comme nous pouvons le constater sur le graphique de coefficient ci-dessus, il y a beaucoup de variables qui ont un coefficient non significatif, ces coefficients n'ont pas beaucoup contribué au modèle et doivent être régularisés voire être éliminés, ce qui éliminera également les variables correspondantes.

Sur graphique de nuage de points, nous allons comparer les valeurs observées sur l'échantillon test et les valeurs prédites sur le même échantillon. Nous réalisons un graphique avec en abscisse les valeurs observées, en ordonnée les valeurs prédites.

```
#graphique
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 7))
plt.scatter(y_test,reg_pred)
plt.plot(numpy.arange(4,50,0.5),numpy.arange(4,50,0.5))
plt.xlabel("Valeurs observées")
plt.ylabel("Valeurs predites")
plt.show()
```

Figure 4 : Lien entre les valeurs observées et prédites



Les points devraient se situer tout au long de la ligne droite lorsque les prédictions sont parfaites. Dans notre cas, la régression est relativement bonne avec un $\mathbf{R^2=0.71}$ obtenu ci-dessus. On note néanmoins que certains points sont très mal modélisés, ils sont très loin de la ligne droite.

Nous allons maintenant aborder la **méthode** de **régularisation**. Cependant, cette **méthode** de régularisation **exige** que les **données** soient **mises** en **échelle**. Dans, notre cas les données sont déjà mises en échelle. Si vous devriez mettre vos données en échelle veuillez cliquer ici pour vous inspirer.

4 Régression Ridge

La régression Ridge ajoute un terme dans la fonction d'erreur des moindres carrés ordinaires qui régularise la valeur des coefficients des variables. Ce terme est la somme des carrés du coefficient multipliée par un paramètre lambda ou alpha ou un autre paramètre si vous préfériez. Le motif de l'ajout de ce terme est de pénaliser la variable correspondante à ce coefficient peu corrélé à la variable cible. Ce terme est appelé norme $\mathbf{L_2}$.

Nous implémentons la régression de Ridge via les lignes de codes ci-dessous en choisissant une valeur aléatoire de pénalisation de coefficients (alpha=1) :

```
# le modèle d'entrainement
ridgeR = Ridge(alpha = 1)
ridgeR.fit(X_train, y_train)
y_pred = ridgeR.predict(X_test)
# Calculer l'erreur quadratique moyenne
mean_squared_error_ridge = np.mean((y_pred - y_test)**2)
print(mean_squared_error_ridge)
#Obtenir les coefficients de Rige et afficher les
ridge_coefficient = pd.DataFrame()
ridge_coefficient["Columns"]= X_train.columns
ridge_coefficient['Coefficient Estimate'] = pd.Series(ridgeR.coef_)
print(ridge_coefficient)
```

Ces lignes de codes nous donnent la sortie suivante :

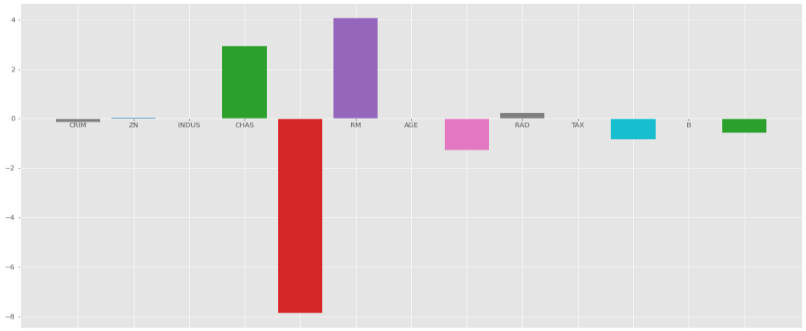
Tableau 6 : Coefficients de détermination et les metrics de performance

```
L'erreur quadratique moyenne :  22.044053089861006
Coefficient de determination: 0.70
    Columns  Coefficient Estimate
0      CRIM              -0.128427
1        ZN               0.036952
2     INDUS               0.017914
3      CHAS               2.932695
4       NOX              -7.848060
5        RM               4.063574
6       AGE              -0.017242
7       DIS              -1.271761
8       RAD               0.225494
9       TAX              -0.009381
10  PTRATIO              -0.827105
11        B               0.011988
12    LSTAT              -0.563474
```

Pour **alpha=1**, nous n'arrivons pas à faire mieux que la **méthode** de **régression linéaire multiple**, car l'**erreur quadratique moyenne** a légèrement augmenté et le **coefficient de détermination** a aussi légèrement baissé. Il faut faire varier la valeur d'alpha pour mieux régulariser. A ce propos, il faut utiliser la **validation croisée** pour trouver un **meilleur alpha** et faire converger les coefficients afin d'améliorer les performances du modèle.

Nous affichons les coefficients du modèle pour une valeur d'alpha égale à 1.

Figure 5 : Visualisation des coefficients de Ridge pour alpha =1



Nous allons utiliser la validation croisée sur différentes valeurs d'alpha et voir si les performances de notre modèle s'améliorent. Nous sélectionnons également la meilleure valeur d'alpha :

```
# Différentes valeurs d'alpha à tester
aphas =[0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1, 0.5, 1, 1,2,3, 4, 5, 6, 7, 8,9
, 10]
# La méthode ridgeCV
ridge_cv=RidgeCV(alphas=alphas, store_cv_values=True)
ridge_mod = ridge_cv.fit(X_train,y_train)
print(ridge_mod.alpha_)
ypred = ridge_mod.predict(X_test)
score = ridge_mod.score(X_test,y_test)
mse = mean_squared_error(y_test, ypred)
print("R2:{0:.3f}, MSE:{1:.2f}, RMSE:{2:.2f}"
    .format(score, mse, np.sqrt(mse)))
# Obtenir les coefficients de Ridge et afficher les
RidgeCV_coefficient = pd.DataFrame()
RidgeCV_coefficient["Columns"]= X_train.columns
RidgeCV_coefficient['Coefficient Estimate'] = pd.Series(ridge_cv.coef_)
print(RidgeCV_coefficient)
```

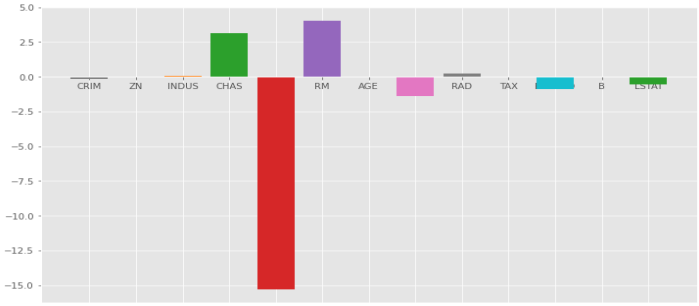
Ces lignes de codes ci-dessous nous donne la sortie suivante :

Tableau 7 : Coefficients du modèle avec la régularisation RidgeCV

```
0.01
R2:0.711, MSE:21.52, RMSE:4.64
    Columns  Coefficient Estimate
0      CRIM              -0.133368
1        ZN               0.035829
2     INDUS               0.048906
3      CHAS               3.117318
4       NOX             -15.270005
5        RM               4.057679
6       AGE              -0.010949
7       DIS              -1.383769
8       RAD               0.242371
9       TAX              -0.008715
10  PTRATIO              -0.909036
11        B               0.011798
12    LSTAT              -0.547399
```

Ce tableau ci-dessous nous indique que le **meilleur alpha** est égal à **0.01** et le **mse** correspondant est égal à **21.54**, quant au **coefficient de détermination**, il est à **71.1%**. Rappelez-vous, ce sont quasiment les metrics retrouvées lors de la régression linéaire multiple. Donc, en utilisant la validation croisée de Ridge, on tombe sur quasiment les mêmes résultats que le modèle de régression multiple. Nous allons visualiser ces coefficients pour nous en assurer.

Figure 6 : Visualisation des coefficients de RidgeCV



A présent, nous allons aborder la régression Lasso, pour voir si on peut encore faire mieux.

NB : Dans la régularisation Ridge, les coefficients ne peuvent jamais être 0. Si on ne les visualise pas bien sur un graphique parce qu'ils sont tout simplement trop petits pour être observés.

5 Régression lasso

La régression au lasso est similaire à la régression de Ridge, sauf que nous ajoutons ici la valeur absolue moyenne des coefficients. Contrairement à la régression Ridge, la régression Lasso peut éliminer complètement la variable non vraiment corrélée à la variable cible en réduisant la valeur du coefficient correspondant à 0.

```
# Le Lasso avec alpha=1
lasso = Lasso(alpha = 1)
lasso.fit(X_train, y_train)
y_pred1 = lasso.predict(X_test)
# Calculer l'erreur quadratique moyenne
mean_squared_error = np.mean((y_pred1 - y_test)**2)
print("L'erreur quadratique moyenne", mean_squared_error)
print("Coefficient de determination: %.2f" % r2_score(y_test, y_pred1))
# Regrouper les coefficients Lasso dans un Dataframe
lasso_coeff = pd.DataFrame()
lasso_coeff["Columns"] = X_train.columns
lasso_coeff['Coefficient Estimate'] = pd.Series(lasso.coef_)
print(lasso_coeff)
```

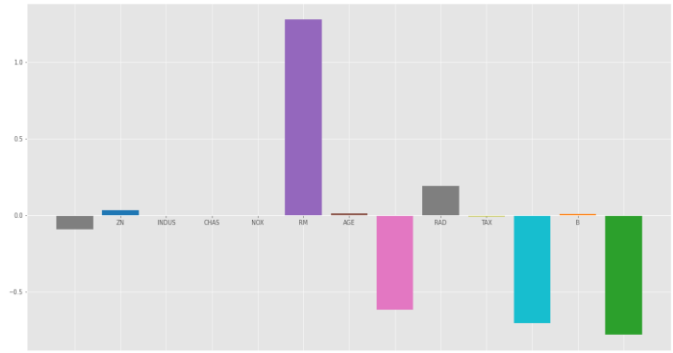
Les lignes de codes ci-dessous nous donnent la sortie suivante :

Tableau 8 : Les coefficients du modèle Lasso

```
L'erreur quadratique moyenne 25.639502928043996
Coefficient de determination: 0.66
    Columns  Coefficient Estimate
0      CRIM             -0.091916
1        ZN              0.034667
2     INDUS             -0.000000
3      CHAS              0.000000
4       NOX             -0.000000
5        RM              1.281317
6       AGE              0.011440
7       DIS             -0.616021
8       RAD              0.191501
9       TAX             -0.009540
10  PTRATIO             -0.703350
11        B              0.010832
12    LSTAT             -0.779921
```

Nous avons utilisé un **alpha égal à 1**, nous conseillons à ce propos d'utiliser la validation croisée pour déterminer le meilleur alpha qui permet d'avoir une faible erreur quadratique moyenne et un coefficient de détermination élevée. Malheureusement, cet alpha ne nous donne pas assez de performance par rapport à la régression linéaire. En effet, on constate que l'**erreur quadratique moyenne** est de **25,64** et le **coefficient de détermination** est de **66%**.

Figure 7 : Visualisation des coefficients de la régularisation au Lasso



Comme expliqué ci-haut d'après le tableau de coefficients et le graphique des coefficients, nous constatons que la régression Lasso a réduit trois coefficients à zéro et les variables correspondantes avec. Donc, nous avons au final 10 variables au lieu de 13. Le modèle lasso ne se performe pas plus que la régression linéaire.

Nous allons aborder dans la suite la méthode **Elastic net** et nous verrons si les performances de notre modèle s'améliorent. Nous finirons par une conclusion après cette dernière partie.

6 Elastic Net

Il s'agit d'une méthode de **régression régularisée** qui combine linéairement les pénalités $\mathbf{L_1}$ et $\mathbf{L_2}$ des méthodes du lasso et Ridge.

L'Elastic-net a été introduit afin de surmonter **deux limitations** du lasso. **Premièrement**, le lasso ne peut sélectionner qu'au plus $\boldsymbol{n}$ variables dans le cas où $\boldsymbol{n < p}$. **Deuxièmement**, en présence d'un groupe de variables fortement corrélées, le lasso ne sélectionne généralement qu'une seule variable du groupe.

Pour introduire la méthode **Elastic net**, nous avons choisi un alpha égal à **0.1** pour éviter le cas où il est égal à **1** ou à **0**, ce qui nous ramènera à la régression Ridge (alpha=0) ou au Lasso (alpha=1).

```
# Instancier et ajuster le modèle
elastic_net = ElasticNet(alpha = 0.1)
elastic_net.fit(X_train, y_train)
# Calculer les prédictions, l'erreur quadratique moyenne et le coéfficient de dé-
termination
y_pred_elasticnet = elastic_net.predict(X_test)
mean_squared_error = np.mean((y_pred_elasticnet - y_test)**2)
print("L'erreur quadratique moyenne", mean_squared_error)
print("Coefficient de determination: %.2f" % r2_score(y_test, y_pred_elasticnet))
# Régrouper les coeffcients ainsi que les variables correspondantes dans un ta-
bleau dataframe
e_net_coeff = pd.DataFrame()
e_net_coeff["Columns"] = X_train.columns
```

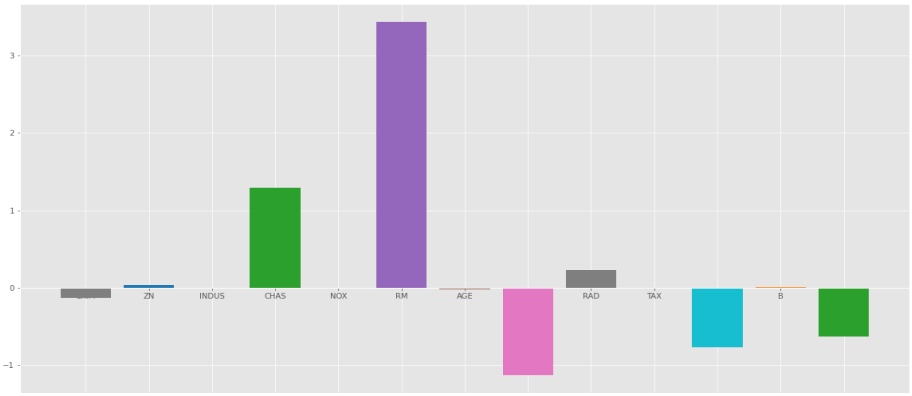
```
e_net_coeff['Coefficient Estimate'] = pd.Series(elastic_net.coef_)
print(elastic_net_coeff)
```

La sortie de ces lignes de codes donne le tableau et le graphique ci-dessous :

Tableau 9 : Coefficients de la méthode de régularisation Elastic Net

```
L'erreur quadratique moyenne 22.9266
Coefficient de determination: 0.69
    Columns  Coefficient Estimate
0      CRIM             -0.126150
1        ZN              0.040934
2     INDUS             -0.010844
3      CHAS              1.294312
4       NOX             -0.000000
5        RM              3.437071
6       AGE             -0.016710
7       DIS             -1.128116
8       RAD              0.233896
9       TAX             -0.011057
10  PTRATIO             -0.769338
11        B              0.012204
12    LSTAT             -0.632976
```

Figure 8 : Visualisation des coefficients d'ElasticNet



En scrutant le tableau et le graphique d'Elastic Net ci-dessus, nous constatons que les metrics d'évaluation (**mse**, **R^2**) ne s'améliorent pas. Alors, nous utiliserons la **validation croisée** sur Elastic Net pour voir si on pourrait faire mieux.

```
# Utiliser ElasticNetCV pour faire varier la valeur d'alpha et d'en selection-
ner le meilleur qui nous permettra peut-être d'obtenir de résultat performant
alphas = [0.0001, 0.001, 0.01, 0.1, 0.3, 0.5, 0.7, 1]
#Instancier le modèle et ajuster le
elastic_cv=ElasticNetCV(alphas=alphas, cv=5)
model = elastic_cv.fit(X_train, y_train)
print(model.alpha_)
#Calculer la prédiction et les metrics d'évaluation
ypred_NetCV= model.predict(X_test)
score = model.score(X_test, y_test)
mse = mean_squared_error(y_test,
ypred_NetCV)
print("R2:{0:.3f}, MSE:{1:.2f}, RMSE:{2:.2f}"
      .format(score, mse, np.sqrt(mse)))
# Stocker les coefficients dans un Dataframe
ElasticNet_coeff = pd.DataFrame()
ElasticNet_coeff["Columns"] = X_train.columns
```

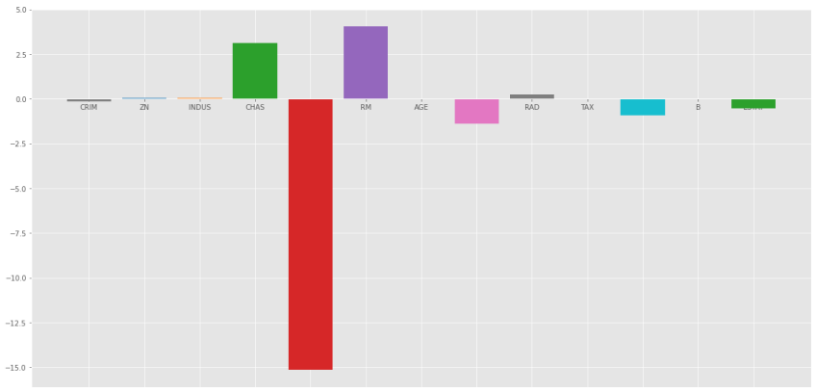
```
ElasticNet_coeff['Coefficient Estimate'] = pd.Series(model.coef_)
print(ElasticNet_coeff)
```

La sortie de ces lignes de codes donne le tableau et le graphique ci-dessous :

Tableau 10 : Coefficients de la méthode de régularisation ElasticNetCV

```
0.0001
R2:0.711, MSE:21.53, RMSE:4.64
    Columns  Coefficient Estimate
0      CRIM             -0.133279
1        ZN              0.035846
2     INDUS              0.048367
3      CHAS              3.114517
4       NOX            -15.141381
5        RM              4.058003
6       AGE             -0.011059
7       DIS             -1.381801
8       RAD              0.242061
9       TAX             -0.008726
10  PTRATIO             -0.907595
11        B              0.011801
12    LSTAT             -0.547660
```

Figure 9 : Visualisation des coefficients d'ElasticNetCV



En scrutant le tableau et le graphique d'ElasticNetCV, nous constatons que le meilleur alpha est de **0,001**, ce qui donne une erreur quadratique moyenne de **21,53** et un coefficient de détermination de **71,1%**. Rappelez-vous, il s'agit quasiment du résultat que nous avons obtenu avec la régression linéaire ou avec la régression Ridge (RidgeCV) en utilisant la validation croisée.

7 Conclusion

La **régression Ridge** ajoute le terme de pénalité de régularisation $\mathbf{L_2}$ à la fonction de perte. Ce terme réduit les coefficients mais ne les rend pas nuls et n'élimine donc pas complètement une variable indépendante. La régression Ridge permet donc de contourner les problèmes de **colinéarité** (**variables explicatives très fortement corrélées entre elles**) dans un contexte où le nombre de variables explicatives en entrée du problème devient de plus en plus élevé. Dans le cadre de ce tutoriel la régression Ridge n'a pas fait mieux que la régression linéaire.

La **régression lasso** ajoute le terme de pénalité de régularisation $\mathbf{L_1}$ à la fonction de perte. Ce terme réduit les coefficients et les rend 0, ce qui élimine complètement la variable indépendante correspondante. Il peut être utilisé pour la sélection de fonctionnalités, etc. Cette méthode n'a pas aussi donnée de bonne performance par rapport à la régression linéaire.

Elastic Net est une combinaison des deux régularisations ci-dessus. Il contient à la fois $\mathbf{L_1}$ et $\mathbf{L_2}$ comme terme de pénalité. Et normalement, il devrait mieux fonctionner que Ridge et Lasso pour la plupart des cas de test. Mais dans notre cas, même Elastic Net n'a pas fait mieux que la méthode de régression linéaire.

Au final, nous constatons que la **régression linéaire multiple** performe mieux que la régularisation dans ce cas, donc nous conseillons d'utiliser la **méthode de sélection pas à pas** ça permettra peut-être d'améliorer plus la performance du modèle. C'est pourquoi le prochain tutoriel sera consacrer aux **méthodes** de **sélection pas à pas** et la **réduction de dimensionnalité**.

TABLE DE MATIERES

1 **Introduction** 1
2 Données 1
3 La régression linéaire multiple 5
4 Régression Ridge 7
5 Régression lasso 9
6 Elastic Net 10
7 Conclusion 13
8 Bibliographie 15

8 Bibliographie

Adrià Serra. Modèles de sélection de sous-ensembles appliqués : https://ichi.pro/fr/modeles-de-selection-de-sous-ensembles-appliques-210120226969166

Implementation of Lasso, Ridge and Elastic Net. Last Updated : 15 May, 2021 : https://www.geeksforgeeks.org/implementation-of-lasso-ridge-and-elastic-net/

Elplatt. Trouver la valeur p (Signification) dans scikit-apprendre la régression linéaire. 2015-01-13 : https://webdevdesigner.com/q/find-p-value-significance-in-scikit-learn-linearregression-11471/

By DataTechNotes. ElasticNet Regression Example in Python. 8/09/2019 : https://www.datatechnotes.com/2019/08/elasticnet-regression-example-in-python.html