1 Introduction

Régression Lasso sous Python. Utilisation du package « scikit-learn ».

Ce tutoriel fait suite au support de cours consacré à la régression régularisée (RAK, 2018). Nous travaillons sous Python avec le package « scikit-learn ».

Au-delà de la simple mise en œuvre de la Régression Lasso, nous effectuons une comparaison avec la régression linéaire multiple usuelle telle qu'elle est proposée dans la librairie « StatsModels » (RAK, 2015) pour montrer son intérêt. Nous verrons entres autres ses apports en termes de sélection de variables et d'optimisation des performances prédictives.

L'exemple est à vocation pédagogique, il s'agit avant tout de décortiquer les mécanismes de l'approche. J'ai par conséquent fait le choix d'utiliser une base de taille réduite (p = 16 variables explicatives) pour que les graphiques soient lisibles (le « Lasso path » par exemple). Dans ce contexte, les propriétés de régularisation de la Régression Lasso ne se démarquent pas vraiment.

2 Données

2.1 Les données « baseball »

Nous traitons le fichier « **Baseball.xlsx** »[1]. L'objectif est d'expliquer le salaire (log du salaire pour être précis) des joueurs à partir de leurs caractéristiques (p = 16 variables), notamment leurs performances en match. Nous disposons de n = 337 observations.

Sous Python, nous importons la première feuille du classeur Excel et nous affichons les propriétés du *dataset*.

```
#changer le répertoire courant
import os
os.chdir("… votre dossier …")

#charger les données
import pandas
```

[1] https://ww2.amstat.org/publications/jse/v6n2/datasets.watnik.html

```
bb = pandas.read_excel("Baseball.xlsx",sheet_name=0)

#description
print(bb.shape) #(337, 17)
print(bb.describe())
```

```
           BatAVG      OnBase        Runs        Hits     Doubles     Triples  \
count  337.000000  337.000000  337.000000  337.000000  337.000000  337.000000
mean     0.257825    0.323973   46.697329   92.833828   16.673591    2.338279
std      0.039546    0.047132   29.020166   51.896322   10.452001    2.543336
min      0.063000    0.063000    0.000000    1.000000    0.000000    0.000000
25%      0.238000    0.297000   22.000000   51.000000    9.000000    0.000000
50%      0.260000    0.323000   41.000000   91.000000   15.000000    2.000000
75%      0.281000    0.354000   69.000000  136.000000   23.000000    3.000000
max      0.457000    0.486000  133.000000  216.000000   49.000000   15.000000

         HomeRuns         RBI       Walks  StrikeOuts  StolenBases  \
count  337.000000  337.000000  337.000000  337.000000   337.000000
mean     9.097923   44.020772   35.017804   56.706231     8.246291
std      9.289934   29.559406   24.842474   33.828784    11.664782
min      0.000000    0.000000    0.000000    1.000000     0.000000
25%      2.000000   21.000000   15.000000   31.000000     1.000000
50%      6.000000   39.000000   30.000000   49.000000     4.000000
75%     15.000000   66.000000   49.000000   78.000000    11.000000
max     44.000000  133.000000  138.000000  175.000000    76.000000

           Errors  FreeAgElig   FreeAge91     ArbElig       Arb91    LNSALARY
count  337.000000  337.000000  337.000000  337.000000  337.000000  337.000000
mean     6.771513    0.397626    0.115727    0.192878    0.029674    6.535579
std      5.927490    0.490135    0.320373    0.395145    0.169938    1.176513
min      0.000000    0.000000    0.000000    0.000000    0.000000    4.690000
25%      3.000000    0.000000    0.000000    0.000000    0.000000    5.440000
50%      5.000000    0.000000    0.000000    0.000000    0.000000    6.610000
75%      9.000000    1.000000    0.000000    0.000000    0.000000    7.670000
max     31.000000    1.000000    1.000000    1.000000    1.000000    8.720000
```

LNSALARY est la variable cible. Elle représente le logarithme népérien de la variable SALARY de la base source accessible en ligne. La distribution de cette dernière était fortement dissymétrique. Le passage au logarithme a permis de la corriger.

2.2 Partition apprentissage-test

Nous partitionnons les données en échantillons d'apprentissage et de test. Les calculs de paramètres des modèles que nous développerons, les différentes optimisations, seront exclusivement effectués sur l'échantillon d'apprentissage, en utilisant la validation croisée lorsque c'est nécessaire. En arbitre impartial, l'échantillon test servira uniquement à évaluer les performances. Nous utiliserons l'erreur quadratique moyenne (MSE : *mean squared error*) pour mesurer les qualités prédictives des modèles.

Nous utilisons l'outil dédié du package « scikit-learn » pour subdiviser les données :

```
#subdivision
from sklearn.model_selection import train_test_split
bbTrain, bbTest = train_test_split(bb,train_size = 200,random_state=69780)

#vérifications
print(bbTrain.shape) #(200, 17)
print(bbTest.shape) #(137, 17)
```

Nous optons pour n^{train} = 200 observations pour l'apprentissage, n^{test} = 137 pour l'évaluation.

3 Régression linéaire multiple

3.1 Modélisation

La régression linéaire multiple est la méthode de référence. Nous exploitons le package « statsmodels » qui propose des outils performants pour la modélisation, l'expertise des modèles et la prédiction (RAK, 2015).

Nous isolons dans une matrice les variables explicatives de l'échantillon d'apprentissage. Nous lui accolons une colonne de valeurs 1 pour indiquer la constante de la régression.

```
#matrice des explicatives
XTrain = bbTrain.iloc[:,:16]
print(XTrain.shape)

#à laquelle est ajoutée (accolée) la constante 1
import statsmodels.api as sm
X1Train = sm.add_constant(XTrain)
```

Vérifions ce qu'il en est sur les premières lignes de la matrice :

```
#vérification
print(X1Train.head())
```

```
     const  BatAVG  OnBase  Runs  Hits  Doubles  Triples  HomeRuns  RBI  \
256    1.0   0.267   0.333    68   130       22        2        20   69
72     1.0   0.211   0.274     5    12        2        0         0    3
89     1.0   0.255   0.321    39   108       22        8         3   26
66     1.0   0.285   0.331    84   170       28       10         8   54
244    1.0   0.265   0.322    44   102       17        1         6   51

     Walks  StrikeOuts  StolenBases  Errors  FreeAgElig  FreeAge91  ArbElig  \
256     45          77           32       4           0          0        1
72       3           2            0       4           0          0        0
89      42          61            2       8           0          0        1
66      42          65           34       5           0          0        1
244     33          56            3      18           1          1        0

     Arb91
256      0
72       0
89       0
66       0
244      0
```

La colonne « const » est bien présente.

Nous isolons le vecteur cible et nous lançons la régression.

```
#vecteur cible
yTrain = bbTrain.iloc[:,16]

#lancer la régression
reg = sm.OLS(yTrain,X1Train)
resReg = reg.fit()
```

Les sorties de « statsmodels » sont assez détaillées.

```
#affichage
print(resReg.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               LNSALARY   R-squared:                       0.799
Model:                            OLS   Adj. R-squared:                  0.781
```

```
Method:                 Least Squares   F-statistic:                     45.38
Date:                Fri, 18 May 2018   Prob (F-statistic):           6.05e-55
Time:                        15:25:33   Log-Likelihood:                -148.19
No. Observations:                 200   AIC:                             330.4
Df Residuals:                     183   BIC:                             386.5
Df Model:                          16
Covariance Type:            nonrobust
==============================================================================
                  coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const           5.4578      0.295     18.487      0.000       4.875       6.040
BatAVG          2.6434      2.793      0.946      0.345      -2.868       8.155
OnBase         -3.4560      2.393     -1.444      0.150      -8.177       1.265
Runs           -0.0090      0.005     -1.653      0.100      -0.020       0.002
Hits            0.0076      0.003      2.397      0.018       0.001       0.014
Doubles        -0.0007      0.009     -0.076      0.939      -0.018       0.017
Triples         0.0112      0.023      0.480      0.632      -0.035       0.057
HomeRuns        0.0157      0.012      1.282      0.202      -0.008       0.040
RBI             0.0051      0.005      1.008      0.315      -0.005       0.015
Walks           0.0107      0.005      2.362      0.019       0.002       0.020
StrikeOuts     -0.0051      0.002     -2.449      0.015      -0.009      -0.001
StolenBases     0.0056      0.004      1.274      0.204      -0.003       0.014
Errors         -0.0167      0.007     -2.245      0.026      -0.031      -0.002
FreeAgElig      1.5934      0.107     14.954      0.000       1.383       1.804
FreeAge91      -0.3648      0.129     -2.830      0.005      -0.619      -0.110
ArbElig         1.2669      0.119     10.634      0.000       1.032       1.502
Arb91           0.3425      0.326      1.052      0.294      -0.300       0.985
==============================================================================
Omnibus:                       14.369   Durbin-Watson:                   2.263
Prob(Omnibus):                  0.001   Jarque-Bera (JB):               33.521
Skew:                          -0.240   Prob(JB):                     5.26e-08
Kurtosis:                       4.947   Cond. No.                     1.45e+04
==============================================================================
```

Le R^2 = 0.799 (coefficient de détermination) : le modèle explique près de 80% de la variabilité de la variable cible. Les 6 explicatives significatives à 5% (p-value < 0.05) dans le modèle sont : Hits, Walks, StrikeOuts, FreeAgElig, FreeAge91 et ArbElig.

Remarque : A ce stade, nous devrions réaliser une sélection de variables (approche fondée sur le F-partiel ou s'appuyant sur l'optimisation des critères AIC / BIC par exemple) avant de procéder à la prédiction. Nous choisissons néanmoins de les conserver toutes dans ce tutoriel pour simplifier la démarche.

3.2 Prédiction

Nous accolons la colonne de valeurs 1 à la matrice des explicatives en test et nous appelons la fonction **predict()** pour opérer la prédiction.

```
#matrice des descripteurs pour échantillon test
XTest = bbTest.iloc[:,:16]
X1Test = sm.add_constant(XTest)

#appliquer le modèle
ypReg = reg.predict(resReg.params,X1Test)
print(ypReg)
```

La lecture des prédictions ne sont pas intéressantes en soi dans notre contexte. Seule nous importe la comparaison avec les valeurs observées sur l'échantillon test. Nous réalisons un graphique avec en abscisse les valeurs observées, en ordonnée les valeurs prédites.

```
 #y obs. sur l'échantillon test
yTest = bbTest.iloc[:,16]

#librairie numpy
import numpy

#graphique
import matplotlib.pyplot as plt
plt.scatter(yTest,ypReg)
plt.plot(numpy.arange(4,10,0.5),numpy.arange(4,10,0.5))
plt.xlabel("Observed values")
plt.ylabel("Predicted values")
plt.show()
```

Les points devraient se situer tout au long de la diagonale principale lorsque les prédictions sont parfaites. Dans notre cas (Figure 1), la régression est relativement bonne. Le R^2=0.799 obtenu ci-dessus le laissait augurer. On note néanmoins que certains points sont très mal modélisés, ils sont très loin de la diagonale.

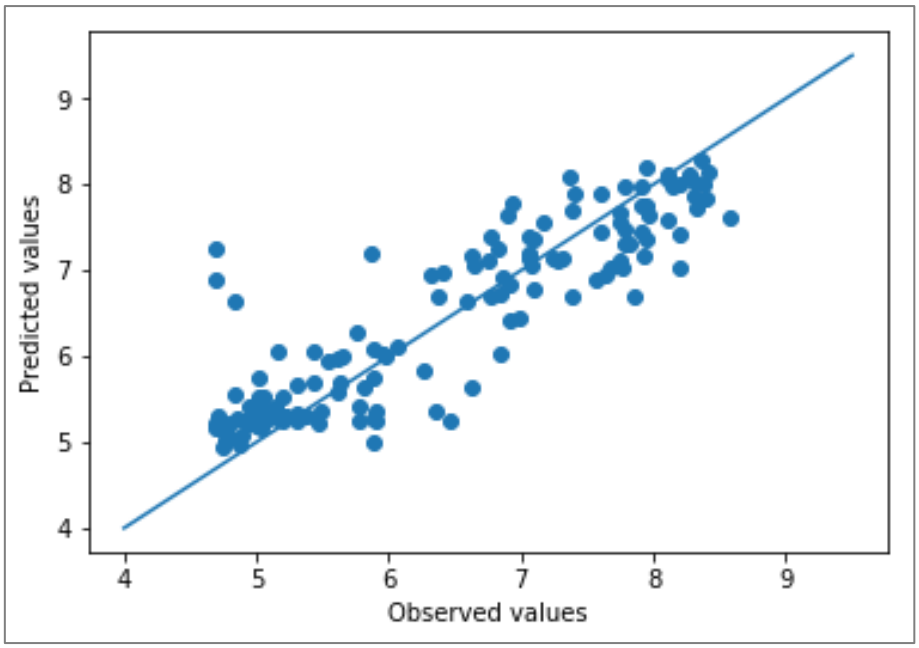

Figure 1 - Observées vs. Prédites - Régression linéaire multiple (statsmodels)

Un graphique est toujours intéressant, un indicateur numérique l'est tout autant, surtout pour comparer les mérites de différents modèles.

Nous calculons la MSE sur l'échantillon test à l'aide l'outil dédié fourni par « scikit-learn » :

```
#mesurer le MSE
from sklearn.metrics import mean_squared_error
print(mean_squared_error(yTest,ypReg)) #0.3352130958741573
```

Nous connaissons la formule,

$$MSE = \frac{1}{n^{test}} \sum_{i=1}^{n^{test}} (y_i - \hat{y}_i)^2$$

Où y_i et $\hat{y}_i$ sont respectivement les valeurs observées et prédites de la variable cible.

Nous pouvons aussi la calculer explicitement :

```
#vérification
print(numpy.mean((yTest-ypReg)**2)) #0.3352130958741573
```

Le résultat est bien le même avec MSE = 0.3352130958741573.

Voyons si nous pouvons faire mieux avec la Régression Lasso…

4 Régression Lasso

4.1 Préparation des données

Il est préférable de centrer et réduire les variables dans la régression régularisée afin que le coefficient de pénalité α (il est noté λ dans le cours, RAK, 2018) agisse de manière homogène sur l'ensemble des coefficients de la régression.

La formule de transformation s'écrit (individu n°i, variable X_j) :

$$z_{ij}^{train} = \frac{x_{ij}^{train} - \bar{x}_j^{train}}{\sigma_j^{train}}$$

Où $\bar{x}_j^{train}$ et σ_j^{train} sont respectivement la moyenne et l'écart-type (la racine carrée de la variance) de la variable X_j, calculées sur l'échantillon d'apprentissage.

```
#centrer et réduire les données d'apprentissage
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
ZTrain =sc.fit_transform(bbTrain)
```

Les variables de la base d'apprentissage sont standardisées, nous disposons des moyennes…

```
#moyennes des variables
print(sc.mean_)
```

```
[2.56955e-01 3.24015e-01 4.69900e+01 9.20400e+01 1.63800e+01 2.32500e+00
 8.92000e+00 4.32900e+01 3.54800e+01 5.47500e+01 8.51500e+00 6.59000e+00
 4.35000e-01 1.45000e-01 1.95000e-01 1.50000e-02 6.60085e+00]
```

… et des variances…

```
#variance des variables
print(sc.var_)
```

```
[1.90247298e-03 2.69822477e-03 8.90979900e+02 2.88477840e+03
 1.11605600e+02 6.37937500e+00 7.89936000e+01 8.65915900e+02
 6.70629600e+02 1.07084750e+03 1.52309775e+02 3.59419000e+01
 2.45775000e-01 1.23975000e-01 1.56975000e-01 1.47750000e-02
 1.28024478e+00]
```

…que nous pourrons réutiliser par la suite.

Les moyennes des variables transformées sont bien nulles (aux erreurs de troncature près) :

```
#affichage des moyennes après transformations
print(numpy.mean(ZTrain,axis=0))
```

```
[-9.72555370e-16  3.10862447e-17 -6.21724894e-17 -8.88178420e-17
  1.06581410e-16 -9.99200722e-17 -4.44089210e-18  5.32907052e-17
  1.24344979e-16 -1.77635684e-17 -8.88178420e-18 -4.44089210e-18
 -2.66453526e-17  1.02140518e-16  4.44089210e-18 -2.22044605e-18
 -2.39808173e-16]
```

Et leurs variances sont égales à 1 :

```
#affichage des variances itou
print(numpy.var(ZTrain,axis=0))
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Ouf ! Le contraire aurait été très ennuyeux.

4.2 Modélisation avec les paramètres par défaut (α = 1.0)

Nous pouvons lancer la régression Lasso de la librairie « scikit-learn ». Nous précisons que la constante n'est pas nécessaire puisque toutes les variables sont centrées, et qu'il n'est pas utile de les normaliser puisqu'elles ont été standardisées en amont. Les autres paramètres sont laissés pas défaut.

```
#régression Lasso, paramètres par défaut (alpha  = 1.0)
from sklearn.linear_model import Lasso
regLasso1 = Lasso(fit_intercept=False,normalize=False)
print(regLasso1)
```

```
Lasso(alpha=1.0, copy_X=True, fit_intercept=False, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False)
```

La valeur par défaut du coefficient de pénalité est **α = 1.0**. Nous ne savons pas vraiment si elle est adaptée à notre jeu de données. Faisons confiance à « scikit-learn » pour l'instant.

Nous lançons la régression sur les données d'apprentissage (centrées et réduites) et nos affichons les coefficients estimés :

```
#apprentissage
regLasso1.fit(ZTrain[:,:16],ZTrain[:,16])

#les coefficients
print(regLasso1.coef_)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Ils sont tous nuls. Difficile de réaliser une prédiction performante avec ça. Manifestement, la valeur ($\alpha = 1.0$) ne convient pas.

Comment peut-on faire pour déterminer une valeur de α qui convient ?

4.3 Lasso Path

Lorsque α est trop élevé, tous les coefficients de la régression sont nuls, nous en avons une illustration ici ; lorsque α est trop faible, proche de 0, nous obtenons les coefficients de la régression linéaire multiple usuelle. Il faut trouver le juste milieu et c'est toute la difficulté de la régression Lasso. L'outil « Lasso path » peut nous y aider. Il produit un graphique qui met en relation les différentes versions de α avec les coefficients estimés. La régression Lasso, contrairement à Ridge, permet de réaliser une sélection de variables en mettant à zéro sélectivement les coefficients. Nous voyons ainsi se dessiner des scénarios de solutions (au sens « différents ensembles de variables sélectionnées ») tout au long du « Lasso path ».

La librairie « scikit-learn » peut nous proposer un jeu de valeurs α à tester. J'ai fait le choix de les spécifier explicitement dans ce tutoriel pour mieux maîtriser le processus.

```
#lasso path (10 valeurs de alpha à tester)
my_alphas = numpy.array([0.001,0.01,0.02,0.025,0.05,0.1,0.25,0.5,0.8,1.0])
```

La fonction **lasso_path()** permet de produire les coefficients estimés correspondants :

```
#obtention des valeurs des coefs. corresp.
from sklearn.linear_model import lasso_path
alpha_for_path, coefs_lasso, _ = lasso_path(ZTrain[:,:16],ZTrain[:,16],alphas=my_alphas)
```

alpha_for_path correspond aux versions de α qui ont été testées, les mêmes que celles indiquées dans le vecteur **my_alphas**, mais rangées différemment.

coefs_lasso est la matrice des coefficients avec 16 lignes (parce que 16 variables explicatives) et 10 colonnes (parce que 10 valeurs différentes de α essayées).

```
#dim. matrice des coefficients
print(coefs_lasso.shape) #(16, 10)
```

Un graphique permet de les mettre en relation :

```
#jeu de couleurs pour faire joli
import matplotlib.cm as cm
couleurs = cm.rainbow(numpy.linspace(0,1,16))

#graphique lasso path (une courbe par variable)
for i in range(coefs_lasso.shape[0]):
    plt.plot(alpha_for_path,coefs_lasso[i,:],c=couleurs[i])

plt.xlabel('Alpha')
plt.ylabel('Coefficients')
plt.title('Lasso path')
plt.show()
```

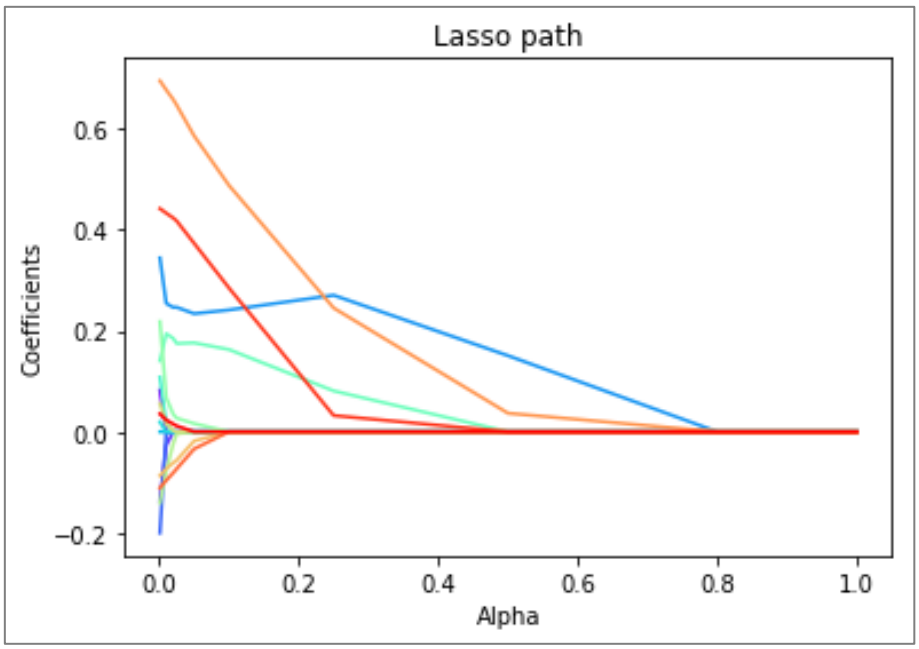

Figure 2 - Lasso path

Pour α = 1.0, tous les coefficients sont nuls effectivement. Pour α = 0.6, deux des coefficients sont non-nuls, ce qui correspond à un scénario où deux seules explicatives sont actives dans la régression. Etc.

De fait, pour chaque valeur de α, nous pouvons identifier le nombre de variables sélectionnées :

```
#nombre de coefs. non-nuls pour chaque alpha
nbNonZero =
numpy.apply_along_axis(func1d=numpy.count_nonzero,arr=coefs_lasso,axis=0)

#affichage mieux organisé alpha vs. nombre de coefs non-nuls
```

```
print(pandas.DataFrame({'alpha':alpha_for_path,'Nb non-zero coefs':nbNonZero}))
   Nb non-zero coefs  alpha
0                  0  1.000
1                  0  0.800
2                  2  0.500
3                  4  0.250
4                  4  0.100
5                  7  0.050
6                  9  0.025
7                 10  0.020
8                 12  0.010
9                 16  0.001
```

Ce tableau est plus engageant sous une forme graphique.

```
#ou sous forme graphique
plt.plot(alpha_for_path,nbNonZero)
plt.xlabel('Alpha')
plt.ylabel('Nb. de variables')
plt.title('Nb. variables vs. Alpha')
plt.show()
```

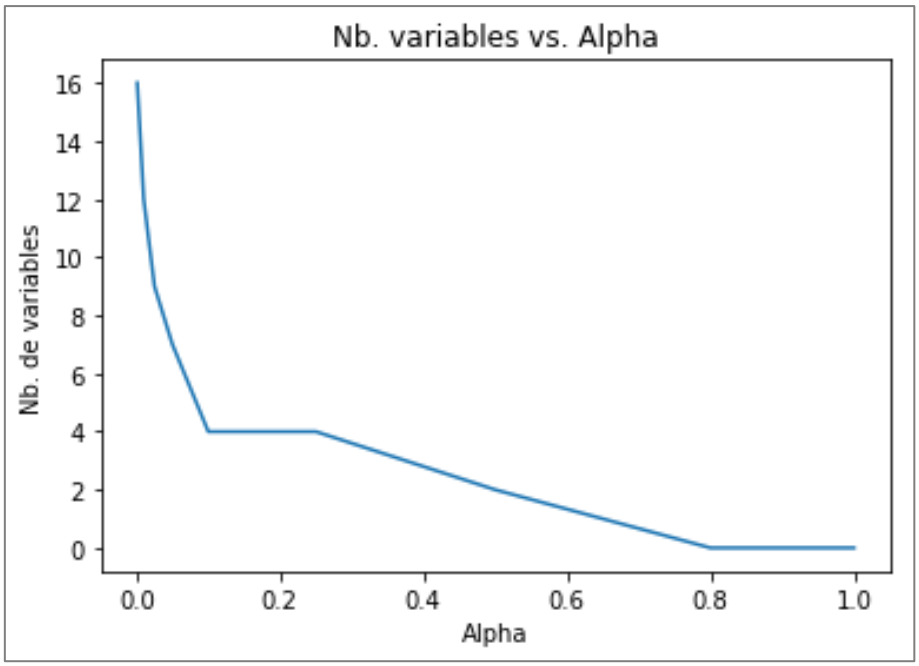

Figure 3 - Nombre de variables sélectionnées dans le modèle en fonction de α

Nous pouvons par exemple identifier les 4 variables sélectionnées pour α = 0.25.

```
#nom des variables
nom_var = bb.columns[:16]

#coefficients pour alpha=0.25 (colonne n°3)
coefs25 = coefs_lasso[:,3]

#affichage des coefficients  pour alpha = 0.25
print(pandas.DataFrame({'Variables':nom_var,'Coefficients':coefs25}))
```

```
    Coefficients    Variables
0       0.000000       BatAVG
1       0.000000       OnBase
2       0.000000         Runs
3       0.270675         Hits
4       0.000000      Doubles
5       0.000000      Triples
6       0.000000     HomeRuns
7       0.081050          RBI
8       0.000000        Walks
9       0.000000   StrikeOuts
10      0.000000  StolenBases
11      0.000000       Errors
12      0.245411   FreeAgElig
13      0.000000    FreeAge91
14      0.032138      ArbElig
15      0.000000        Arb91
```

Il s'agit de Hits, RBI, FreeAgElig et ArbElig.

4.4 Optimisation en validation croisée

Nous disposons de scénarios de solutions pour différentes versions de α. Mais nous ne savons pas laquelle est la plus performante en prédiction. Dans cette section, nous utilisons la validation croisée **LassoCV()** pour l'identifier.

```
#outil pour la détection de la solution la plus performante en validation croisée
#random_state = 0 pour fixer l'initialisation du générateur de nombre aléatoire
#cv = 5 pour 5-fold validation croisée
lcv = LassoCV(alphas=my_alphas,normalize=False,fit_intercept=False,random_state=0,cv=5)

#lancement sur l'échantillon d'apprentissage
lcv.fit(ZTrain[:,:16],ZTrain[:,16])
```

```
#valeurs des alphas qui ont été testés
print(lcv.alphas_) #[1.    0.8   0.5   0.25  0.1   0.05  0.025 0.02  0.01  0.001]

#valeurs des MSE en validation croisée
print(lcv.mse_path_)
```

```
[[0.93649681 0.8999204  1.10600324 1.0127433  1.04483625]
 [0.93649681 0.8999204  1.10600324 1.0127433  1.04483625]
 [0.74836709 0.6781293  0.88481882 0.83005599 0.83769334]
 [0.49603196 0.33392406 0.54118262 0.42756911 0.5835088 ]
 [0.2454802  0.22685436 0.29207343 0.22519332 0.41121256]
 [0.19487357 0.22261075 0.25240979 0.19031063 0.376714  ]
 [0.16429895 0.2193215  0.24264611 0.1854277  0.35736191]
 [0.15917131 0.22139796 0.24283309 0.18591439 0.35482339]
 [0.14588651 0.22529395 0.24602707 0.1881818  0.3489667 ]
 [0.13990895 0.23184779 0.25638676 0.2047393  0.34858521]]
```

La propriété **mse_path_** est une matrice (10 x 5) : 10 parce que 10 versions de α ont été testées ; 5 parce que nous avons demandé une 5-fold validation croisée.

Nous calculons la moyenne pour disposer d'une mesure de performance synthétique pour chaque scénario. Puis nous affichons le tableau mettant en relation α et le MSE (moyen) en validation croisée.

```
#moyenne mse en validation croisée pour chaque alpha
avg_mse = numpy.mean(lcv.mse_path_,axis=1)

#alphas vs. MSE en cross-validation
print(pandas.DataFrame({'alpha':lcv.alphas_,'MSE':avg_mse}))
```

```
        MSE  alpha
0  1.000000  1.000
1  1.000000  0.800
2  0.795813  0.500
3  0.476443  0.250
4  0.280163  0.100
5  0.247384  0.050
6  0.233811  0.025
7  0.232828  0.020
8  0.230871  0.010
9  0.236294  0.001
```

Que nous pouvons également exprimer sous forme graphique :

```
#sous-forme graphique
plt.plot(lcv.alphas_,avg_mse)
plt.xlabel('Alpha')
plt.ylabel('MSE')
plt.title('MSE vs. Alpha')
plt.show()
```

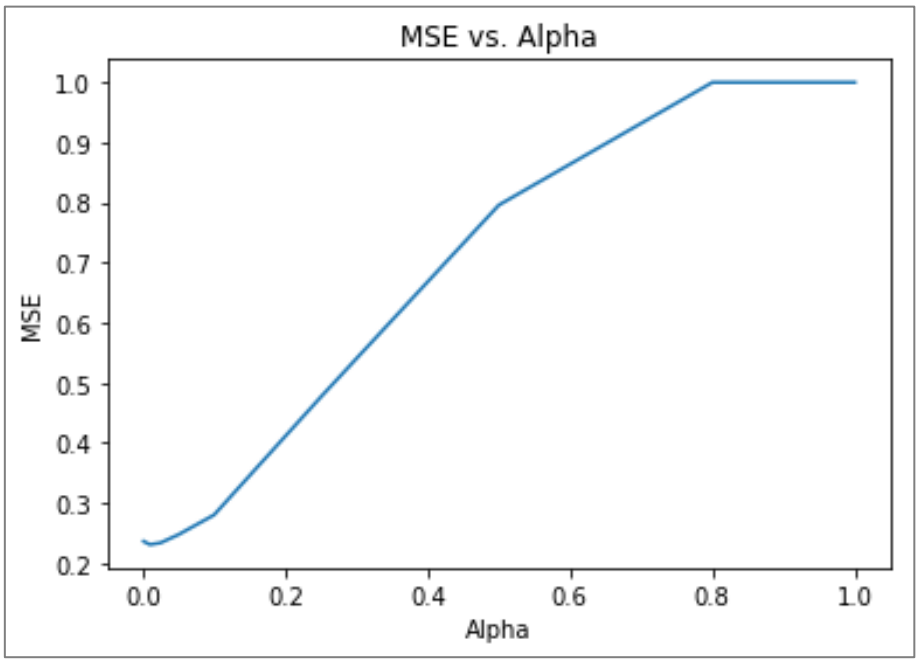

Figure 4 - Performances prédictives (MSE) en fonction de α

α = 0.01 est la solution qui minimise le MSE (0.230871), elle est composée de 12 variables explicatives c.-à-d. 12 coefficients sont différents de zéro dans le modèle (Figure 3).

```
#best alpha
print(lcv.alpha_) #0.01
```

4.5 Evaluation sur l'échantillon test

Pour appliquer le modèle optimal (α = 0.01), nous devons tout d'abord centrer et réduire les variables explicatives de l'échantillon test en utilisant les paramètres (moyennes, écarts-type) calculés sur l'échantillon d'apprentissage c.-à-d.

$$z_{ij}^{test} = \frac{x_{ij}^{test} - \bar{x}_j^{train}}{\sigma_j^{train}}$$

C'est ce que fait la fonction **transform()** de l'objet StandardScaler

```
#transformation des variables des données test
ZTest = sc.transform(bbTest)
```

Nous pouvons maintenant appliquer le modèle.

```
#prediction avec ce modèle
ypzLasso = lcv.predict(ZTest[:,:16])
```

Nous disposons d'une prédiction standardisée de la variable à prédire, il faut la dé-standardiser pour qu'elle soit exprimée dans l'unité originelle. Ici aussi, nous devons utiliser les paramètres calculés sur l'échantillon d'apprentissage :

$$\hat{y}_i^{test} = \hat{z}_{iy}^{test} \times \sigma_y^{train} + \bar{y}^{train}$$

Où $\hat{z}_{iy}^{test}$ est la prédiction de la régression Lasso sur les variables centrées et réduites.

```
#dé-standardization de la prédiction, [-1] parce que y est en dernière position
ypLasso = ypzLasso*numpy.sqrt(sc.var_[-1]) + sc.mean_[-1]
```

Nous pouvons dès lors confronter ces valeurs prédites avec les valeurs observées sur l'échantillon test

```
#performances en prédiction
print(mean_squared_error(yTest,ypLasso)) #0.31204589721406967
```

Nous obtenons une MSE = 0.31204589721406967. Très légèrement moindre (plus la MSE est faible, meilleur est le modèle) qu'avec la régression usuelle avec « statsmodels ».

Remarque : J'ai utilisé LassoCV qui est un outil d'optimisation dédié spécifiquement à la régression Lasso. Mais j'aurais pu également utiliser l'outil plus générique GridSearchCV. Ce dernier est plus générique et convient à plusieurs algorithmes de machine learning. Le premier est plus spécifique disais-je, il peut notamment produire pour nous un jeu de valeurs α à explorer si nous ne savons pas les préciser explicitement. C'est un avantage certain si nous n'avons aucune idée des valeurs à essayer.

5 Conclusion

« Tout ça pour ça serait-on tenté de dire », l'écart de performances entre la régression usuelle et lasso n'est pas ce que l'on peut qualifier de mirobolant sur nos données. C'est la démarche

qui est importante. Il faut être très attentif au rôle du coefficient de pénalité α dans la régression Lasso. Ainsi, la valeur par défaut de « scikit-learn » (α = 1.0) n'était absolument pas adaptée à notre *dataset*. Dans ce tutoriel, nous l'avons mis en relation avec le nombre de variables sélectionnées d'une part, avec les performances prédictives via la MSE mesurée en validation croisée d'autre part. Le processus de standardisation / dé-standardisation des variables pour l'apprentissage et la prédiction était également un aspect important.

Sur notre base, avec un ratio nombre de variables explicatives (p) et nombre d'observations (n) relativement confortable (parce que faible), la régression Lasso ne pouvait pas vraiment se distinguer. Un prolongement naturel de ce tutoriel serait de travailler sur des bases où p est largement plus grand que n. Dans cette configuration, la régression linéaire n'est pas possible (avec les implémentations usuelles) ou très instable (si on utilise des approches de type gradient stochastique) (RAK, 2018b). Les propriétés de régularisation de Lasso (et plus généralement Elasticnet) font alors merveille. Nous pouvons utiliser la même démarche pour choisir la valeur optimale du paramètre α. A priori, nous devrions fixer / obtenir une valeur (de α) plus élevée pour mieux « lisser » l'apprentissage et contrer la surdimensionnalité.

6 Références

- (RAK, 2015) « Python – Econométrie avec StatsModels », Septembre 2015 ; http://tutoriels-data-mining.blogspot.fr/2015/09/python-econometrie-avec-statsmodels.html
- (RAK, 2018) « Ridge – Lasso - Elasticnet », Mai 2018 ; http://tutoriels-data-mining.blogspot.fr/2018/05/ridge-lasso-elasticnet.html
- (RAK, 2018b) « Descente de gradient stochastique sous Python », Mai 2018 ; http://tutoriels-data-mining.blogspot.fr/2018/05/descente-de-gradient-stochastique-sous.html
- Scikit-learn, « Generalized Linear Models », Section 1.1 ; http://scikit-learn.org/stable/modules/linear_model.html