1 Introduction

Pratique de la régression logistique sous Python via les packages « statsmodels » et « scikit-learn ». Estimation des coefficients, inférence statistique, évaluation du modèle, en resubstitution et en test, mesure des performances prédictives, courbe ROC, critère AUC.

Ce tutoriel fait suite à la série d'exercices corrigés de régression logistique sous R (TD 1 à TD 8). Il aurait pu constituer la 9ème séance des travaux dirigés où l'on ferait travailler les étudiants sous Python. J'aime bien alterner les logiciels dans ma pratique de l'enseignement. J'ai quand-même préféré le format tutoriel parce qu'il y a de nombreux commentaires à faire sur le mode opératoire des outils que nous utiliserons. Les librairies « statsmodels » et « scikit-learn » adoptent des points de vue très différents sur les mêmes traitements. Il est important de mettre en relation directe les thèmes et les commandes avec le cours rédigé disponible en ligne (LIVRE, mai 2017).

Enfin, ce document peut être vu comme le pendant pour la régression logistique de celui consacré à la régression linéaire disponible sur notre site (« Econométrie avec StatsModels », septembre 2015).

2 Données

Nous utilisons les données sur les infidélités (TD 2.b) à 2 modifications près : la variable cible « Infidélité » a été transformée en binaire 0/1 ; le dataset a été partitionné en amont en échantillons d'apprentissage (401 observations) et de test (200), il y a par conséquent deux feuilles dans le classeur Excel « **infidelites_python.xlsx** ».

2.1 Importation de l'échantillon d'apprentissage

Nous chargeons la première feuille « **train** » de notre classeur Excel et nous en inspectons le contenu.

```
#changement de répertoire
import os
os.chdir("... votre dossier de travail...")

#chargement des données d'apprentissage
import pandas
DTrain = pandas.read_excel("infidelites_python.xlsx", sheet_name = "train")

#info
print(DTrain.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 401 entries, 0 to 400
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Sex             401 non-null    int64
 1   Age             401 non-null    float64
 2   YearsMarried    401 non-null    float64
 3   Children        401 non-null    int64
 4   Religious       401 non-null    int64
 5   Education       401 non-null    int64
 6   Occupation      401 non-null    int64
 7   RatingMarriage  401 non-null    int64
 8   Infidelite      401 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 28.3 KB
```

Nous isolons la variable cible (Y) et les explicatives potentielles (X) dans deux structures distinctes. Nous comptabilisons les effectifs par classe. Nous avons 295 personnes fidèles et 106 prévaricateurs dans notre échantillon.

```
#y (Infidelite) est la dernière colonne
yTrain = DTrain.iloc[:,-1]

#X (les autres) sont les variables qui précèdent la dernière
XTrain = DTrain.iloc[:,:-1]

#comptage des modalités de y
print(yTrain.value_counts())
```

```
0    295
1    106
Name: Infidelite, dtype: int64
```

2.2 Importation et préparation des données test

Nous réalisons les mêmes étapes sur notre échantillon « **test** ». Nous chargeons les données, …

```
#chargement des données test
DTest = pandas.read_excel("infidelites_python.xlsx", sheet_name = "test")

#info
print(DTest.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Sex             200 non-null    int64
 1   Age             200 non-null    float64
 2   YearsMarried    200 non-null    float64
 3   Children        200 non-null    int64
 4   Religious       200 non-null    int64
```

```
 5   Education       200 non-null    int64
 6   Occupation      200 non-null    int64
 7   RatingMarriage  200 non-null    int64
 8   Infidelite      200 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 14.2 KB
```

… puis nous préparons les structures.

```
#y (Infidelite) est la dernière colonne
yTest = DTest.iloc[:,-1]

#X (les autres) sont les variables qui précèdent la dernière
XTest = DTest.iloc[:,:-1]

#comptage des modalités de y
print(yTest.value_counts())
```

```
0    156
1     44
Name: Infidelite, dtype: int64
```

3 Régression logistique avec "statsmodels"

J'avais longuement étudié le package « statsmodels » dans un précédent tutoriel consacré à la régression linéaire (septembre 2015). J'avais noté à cette occasion qu'il était marqué par une forte culture statistique et correspondait parfaitement à l'usage que je pouvais en faire dans mon cours d'économétrie. Nous ferons le même constat dans cette étude. Il propose la majorité des outils nécessaires pour réaliser une inférence statistique efficace à partir des résultats de l'estimation des paramètres sur les données.

Première vérification incontournable s'agissant des packages pour Python, nous affichons tout d'abord le numéro de version que nous utiliserons.

```
#importation de la librairie de calcul
import statsmodels as sm

#vérification de version
print(sm.__version__)
```

```
0.11.1
```

3.1 Modélisation et inspection des résultats

3.1.1 Ajouter une constante à la matrice de données

Il existe différentes manières d'initier une analyse sous « statsmodels ». Il est possible notamment de définir la régression via une formule comme sous R. Mais j'ai opté pour le passage des variables

dépendantes et indépendantes par des vecteurs et matrices pour des motifs de cohérence. Ainsi, nous procéderons à des manipulations identiques sur les échantillons d'apprentissage et de test. Avec ce mode opératoire, il faut ajouter explicitement une constante à la matrice des variables explicatives.

```
#importation de l'outil
from statsmodels.tools import add_constant

#données X avec la constante
XTrainBis = sm.tools.add_constant(XTrain)

#vérifier la structure
print(XTrainBis.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 401 entries, 0 to 400
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   const           401 non-null    float64
 1   Sex             401 non-null    int64
 2   Age             401 non-null    float64
 3   YearsMarried    401 non-null    float64
 4   Children        401 non-null    int64
 5   Religious       401 non-null    int64
 6   Education       401 non-null    int64
 7   Occupation      401 non-null    int64
 8   RatingMarriage  401 non-null    int64
dtypes: float64(3), int64(6)
memory usage: 28.3 KB
```

Une colonne « const » de valeur 1 est insérée dans la première colonne. Nous visualisons les premières lignes de la structure pour bien comprendre la nouvelle configuration de la matrice des explicatives.

```
#visualisation des premières lignes de la structure
#premières lignes
print(XTrainBis.head())
```

```
   const  Sex   Age  YearsMarried  Children  Religious  Education  Occupation  \
0    1.0    0  22.0          0.75         0          2         18           6
1    1.0    0  32.0         15.00         1          3         14           1
2    1.0    1  27.0          4.00         1          4         20           5
3    1.0    0  22.0          1.50         0          2         16           4
4    1.0    0  27.0          7.00         1          3         14           1

   RatingMarriage
0               5
1               2
2               5
3               5
4               4
```

3.1.2 Modélisation Logit

Nous pouvons lancer la régression maintenant. Après appel du constructeur de la classe `Logit()` où nous passons les données, nous faisons appel à la fonction `fit()` qui génère un objet résultat doté de propriétés et méthodes qui nous seront très utiles par la suite.

```
#importation de la classe de calcul
from statsmodels.api import Logit

#régression logistique - on passe la cible et les explicatives
lr = Logit(endog=yTrain,exog=XTrainBis)

#lancer les calculs
#algorithme de Newton-Raphson utilisé par défaut
#https://www.statsmodels.org/stable/generated/statsmodels.discrete.discrete_model.Logit.fit.html
res = lr.fit()
```

Par défaut, l'outil s'appuie sur l'algorithme de Newton-Raphson (LIVRE, section 1.5). D'autres procédures d'optimisation sont disponibles. 6 itérations ont été nécessaires pour maximiser la log-vraisemblance (LIVRE, section 1.4, équation 1.8).

```
Optimization terminated successfully.
         Current function value: 0.517885
         Iterations 6
```

Nous affichons ensuite les propriétés de l'objet produit par `fit()`.

```
#propriétés de l'objet résultat
#https://www.statsmodels.org/stable/generated/statsmodels.discrete.discrete_model.LogitResults.html
print(dir(res))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_cache', '_data_attr',
'_get_endog_name', '_get_robustcov_results', '_use_t', 'aic', 'bic', 'bse',
'conf_int', 'cov_kwds', 'cov_params', 'cov_type', 'df_model', 'df_resid', 'f_test',
'fittedvalues', 'get_margeff', 'initialize', 'k_constant', 'llf', 'llnull', 'llr',
'llr_pvalue', 'load', 'mle_retvals', 'mle_settings', 'model', 'nobs',
'normalized_cov_params', 'params', 'pred_table', 'predict', 'prsquared', 'pvalues',
'remove_data', 'resid_dev', 'resid_generalized', 'resid_pearson', 'resid_response',
'save', 'scale', 'set_null_options', 'summary', 'summary2', 't_test',
't_test_pairwise', 'tvalues', 'use_t', 'wald_test', 'wald_test_terms']
```

Les résultats détaillés de la régression sont affichés avec la commande `summary()`.

```
#résumé des résultats
print(res.summary())
```

```
                           Logit Regression Results
==============================================================================
Dep. Variable:             Infidelite   No. Observations:                  401
Model:                          Logit   Df Residuals:                      392
Method:                           MLE   Df Model:                            8
Date:                Mon, 30 Mar 2020   Pseudo R-squ.:                  0.1033
Time:                        17:59:35   Log-Likelihood:                -207.67
converged:                       True   LL-Null:                       -231.60
Covariance Type:            nonrobust   LLR p-value:                 1.056e-07
==============================================================================
                     coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const              1.4653      1.064      1.378      0.168      -0.619       3.550
Sex                0.3855      0.294      1.314      0.189      -0.190       0.961
Age               -0.0572      0.022     -2.551      0.011      -0.101      -0.013
YearsMarried       0.0995      0.038      2.596      0.009       0.024       0.175
Children           0.7662      0.382      2.006      0.045       0.018       1.515
Religious         -0.2580      0.109     -2.360      0.018      -0.472      -0.044
Education          0.0113      0.059      0.190      0.849      -0.105       0.127
Occupation         0.0263      0.086      0.306      0.759      -0.142       0.194
RatingMarriage    -0.4684      0.108     -4.322      0.000      -0.681      -0.256
==============================================================================
```

Les résultats sont tout à fait conformes à ce que l'on pourrait attendre de ce type d'outil. Dans la partie haute, nous y reviendrons, nous disposons des informations globales sur la modélisation. Dans la partie basse, nous visualisons les coefficients estimés et les éléments pour l'inférence statistique (écarts-type estimés des coefficients, statistique de test de significativité, p-value du test de significativité, intervalle de confiance des coefficients à 95%).

Par comparaison, voici les sorties de Tanagra sur les mêmes données (Figure 1).

Adjustement quality

Predicted attribute		Infidelite
Positive value		yes
Number of examples		401
Model Fit Statistics		
Criterion	Intercept	Model
AIC	465.193	433.344
SC	469.186	469.29
-2LL	463.193	415.344
Model Chi test (LR)		
Chi-2		47.8486
d.f.		8
P(>Chi-2)		0
R-like		
McFadden's R		0.1033
Cox and Snell's R		0.1125
Nagelkerke's R		0.1642

Attributes in the equation

Attribute	Coef.	Std-dev	Wald	Signif
constant	1.465329	1.0636	1.8979	0.1683
Sex	0.385546	0.2935	1.7256	0.189
Age	-0.057159	0.0224	6.506	0.0108
YearsMarried	0.099487	0.0383	6.7378	0.0094
Children	0.766212	0.3819	4.0247	0.0448
Religious	-0.258015	0.1093	5.5691	0.0183
Education	0.011273	0.0593	0.0362	0.8492
Occupation	0.026285	0.0858	0.0939	0.7593
RatingMarriage	-0.468423	0.1084	18.6761	0

Figure 1 - Sorties de Tanagra sur les données Infidélités (401 obs. en apprentissage)

Des calculs intermédiaires sont possibles relativement simplement. Par exemple, si nous souhaitons obtenir les intervalles de confiance à 90% des coefficients, nous faisons appel à la méthode `conf_int()` de l'objet résultat fourni par `fit()`.

```
#intervalle de confiance des coefficients à 90%
print(res.conf_int(alpha=0.1))
```

```
                    0         1
const          -0.284213  3.214872
Sex            -0.097223  0.868315
Age            -0.094019 -0.020299
YearsMarried    0.036444  0.162529
Children        0.137998  1.394426
```

```
Religious      -0.437853 -0.078178
Education      -0.086216  0.108762
Occupation     -0.114812  0.167382
RatingMarriage -0.646711 -0.290135
```

3.2 Evaluation globale du modèle en resubstitution

L'évaluation en resubstitution consiste à utiliser les données d'apprentissage même pour mesurer la qualité du modèle : en termes de performances prédictives, nous savons dans ce cas que les indicateurs calculés sont souvent trop optimistes ; en termes d'approximation des probabilités d'appartenance aux classes, ce qui donne lieu à des tests statistiques usuels dans la pratique de la régression logistique mais peu connus en machine learning.

3.2.1 Matrice de confusion en resubstitution

La matrice de confusion résulte de la confrontation entre les classes observées (valeurs de la valeur cible) et estimées à l'aide du modèle (LIVRE, section 2.1). La propriété (.fittedvalues) correspond aux valeurs du LOGIT (LIVRE, section 1.3) calculées sur les données d'apprentissage.

```
#valeurs estimées par la régression en resubstitution
print(res.fittedvalues)
```

```
0     -2.215082
1      0.367956
2     -1.545559
3     -2.215584
4     -1.078986
         ...
396   -1.275453
397   -0.038346
398   -3.119919
399   -1.826017
400   -2.575001
Length: 401, dtype: float64
```

Nous pouvons reproduire ces valeurs à partir des coefficients de la régression et de la description des observations. Pour le premier individu (n°0) par exemple :

```
#Voici les coefficients estimés
print(res.params)
```

```
const            1.465329
Sex              0.385546
Age             -0.057159
YearsMarried     0.099487
Children         0.766212
Religious       -0.258015
Education        0.011273
Occupation       0.026285
```

```
RatingMarriage   -0.468423
dtype: float64
```

```
#voici la description du premier individu
print(XTrainBis.iloc[0,:])
```

```
const              1.00
Sex                0.00
Age               22.00
YearsMarried       0.75
Children           0.00
Religious          2.00
Education         18.00
Occupation         6.00
RatingMarriage     5.00
Name: 0, dtype: float64
```

```
#et si on fait le produit scalaire – valeur du LOGIT pour l'individu n°0
import numpy
print(numpy.sum(res.params*XTrainBis.iloc[0,:]))
```

```
-2.215082315066873
```

Nous pouvons déduire du LOGIT la prédiction en utilisant une règle d'affectation simple (LIVRE, section 1.3).

```
#la règle d'affectation consiste à confronter le LOGIT à la valeur seuil 0
predResub = numpy.where(res.fittedvalues > 0, 1, 0)
print(predResub)
```

```
[0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
```

Via un tableau croisé entre les classes observées et prédites, nous obtenons la matrice de confusion.

```
#on peut en déduire la matrice de confusion
print(pandas.crosstab(yTrain,predResub))
```

```
col_0         0   1
Infidelite
0           284  11
1            87  19
```

Laquelle peut être obtenue directement via la fonction .pred_table() de l'objet régression.

```
#matrice de confusion en resubstitution directement fournie par l'outil
print(res.pred_table())
```

```
[[284.  11.]
 [ 87.  19.]]
```

Plusieurs indicateurs peuvent être extraits de cette matrice (LIVRE, section 2.1.1). Nous y reviendrons plus loin lorsqu'elle sera construite à partir de l'échantillon test. Les valeurs obtenues seront autrement plus représentatives des performances réelles du modèle.

3.3 Pseudo-R2

Les pseudo-R² sont des indicateurs similaires dans l'esprit au R² de la régression linéaire. Il sont basés sur la confrontation entre la log-vraisemblance du modèle et de celle du modèle trivial composé uniquement de la constante (null model) (LIVRE, section 1.6 ; Tableau 1.1.).

- **R² de McFadden.**

```
#accès à la log-vraisemblance du modèle
print("Log-vraisemblance du modèle : %.4f" % (res.llf))

#log-vraisemblance du null modèle
print("Log-vraisemblance du null modèle : %.4f" % (res.llnull))
```

```
Log-vraisemblance du modèle : -207.6719
Log-vraisemblance du null modèle : -231.5963
```

```
#R2 de McFadden
R2MF = 1 - res.llf / res.llnull
print(R2MF)
```

```
0.10330179930318728
```

```
#qui est founi directement par l'outil
print(res.prsquared)
```

```
0.10330179930318728
```

- **R² de Cox et Snell.**

```
#exponenielle de LL_null
L0 = numpy.exp(res.llnull)

#exponentielle de LL_modèle
La = numpy.exp(res.llf)

#taille de l'échantillon
n = DTrain.shape[0]

#R2 de Cox et Snell
R2CS = 1.0 - (L0 / La)**(2.0/n)
print("R2 de Cox - Snell : %.4f" % (R2CS))
```

```
R2 de Cox - Snell : 0.1125
```

– **R^2 de Nagelkerke.**

```
#max du R2 de COx-Snell
maxR2CS = 1.0 - (L0)**(2.0/n)

#R2 de Nagelkerke
R2N = R2CS / maxR2CS
print("R2 de Nagelkerke : %.4f" % (R2N))
```

```
R2 de Nagelkerke : 0.1642
```

3.4 Evaluation basée sur les scores

L'évaluation basée sur les scores analyse dans quelle mesure les probabilités d'affectation à la modalité cible (Y = 1) fournis par le modèle, que l'on nommera « scores » dans ce qui suit, sont de qualité satisfaisante. Nous pouvons les calculer en appliquant la fonction logistique (.cdf) sur le LOGIT (LIVRE, section 1.3, équation 1.5).

```
#scores fournis par la régression
scores = lr.cdf(res.fittedvalues)
print(scores[:10])
```

```
[0.09840424 0.59096505 0.17572855 0.09835976 0.25369795 0.21706862
 0.05574685 0.38659419 0.35418794 0.1773004 ]
```

```
#vérifions la première valeur (individu n°0)
s0 = 1.0/(1.0 + numpy.exp(-1.0 * res.fittedvalues[0]))
print("Score du 1er individu %.4f" % (s0))
```

```
Score du 1er individu 0.0984
```

3.4.1 Diagramme de fiabilité

Le diagramme de fiabilité est un outil de diagnostic graphique. Il oppose les scores estimés par le modèle aux « scores observés ». Ces derniers sont obtenus en calculant la proportion des positifs observés dans des groupes d'observations. Lesquels groupes sont constitués à partir d'un découpage en intervalles de largeurs égales des probabilités d'affectation (LIVRE, section 2.2).

```
#data frame temporaire avec y et les scores
df = pandas.DataFrame({"y":yTrain,"score":scores})

#5 intervalles de largeur égales
intv = pandas.cut(df.score,bins=5,include_lowest=True)

#intégrées dans le df
df['intv'] = intv
print(df)
```

```
     y     score            intv
0    0  0.098404   (0.028, 0.17]
1    1  0.590965  (0.589, 0.729]
2    0  0.175729   (0.17, 0.309]
```

```
3    0  0.098360   (0.028, 0.17]
4    0  0.253698   (0.17, 0.309]
..  ..       ...             ...
396  0  0.218325   (0.17, 0.309]
397  1  0.490415  (0.449, 0.589]
398  0  0.042293   (0.028, 0.17]
399  0  0.138713   (0.028, 0.17]
400  0  0.070765   (0.028, 0.17]

[401 rows x 3 columns]
```

A partir de ce dataset, nous pouvons calculer la moyenne des scores estimés dans chaque groupe délimité par les intervalles.

```
#moyenne des scores par groupe
m_score = df.pivot_table(index="intv",values="score",aggfunc="mean")
print(m_score)
```

```
                  score
intv
(0.028, 0.17]   0.107124
(0.17, 0.309]   0.233010
(0.309, 0.449]  0.375960
(0.449, 0.589]  0.499698
(0.589, 0.729]  0.663811
```

Puis la proportion des observations (scores observés) positives dans les mêmes groupes. La variable cible Y étant binaire (0/1), la moyenne fait très bien l'affaire.

```
#moyenne des y - qui équivaut à une proportion puisque 0/1
m_y = df.pivot_table(index="intv",values="y",aggfunc="mean")
print(m_y)
```

```
                       y
intv
(0.028, 0.17]   0.125000
(0.17, 0.309]   0.188312
(0.309, 0.449]  0.414286
(0.449, 0.589]  0.571429
(0.589, 0.729]  0.600000
```

Le diagramme de fiabilité (*reliability diagram*) est un graphique nuage de points opposant les scores estimés et observés. S'ils forment une droite, nous pouvons considérer que la modélisation est pertinente car le modèle arrive à approcher de manière satisfaisante l'appartenance aux classes des individus.

```
#pouvoir faire apparaître le graphique dans le notebook
%matplotlib inline

#insérer le graphique
import matplotlib.pyplot as plt

#construire la diagonale
```

```
plt.plot(numpy.arange(0,1,0.1),numpy.arange(0,1,0.1),'b')

#rajouter notre diagramme
plt.plot(m_score,m_y,"go-")

#titre
plt.title("Diagramme de fiabilité")

#faire apparaître
plt.show()
```

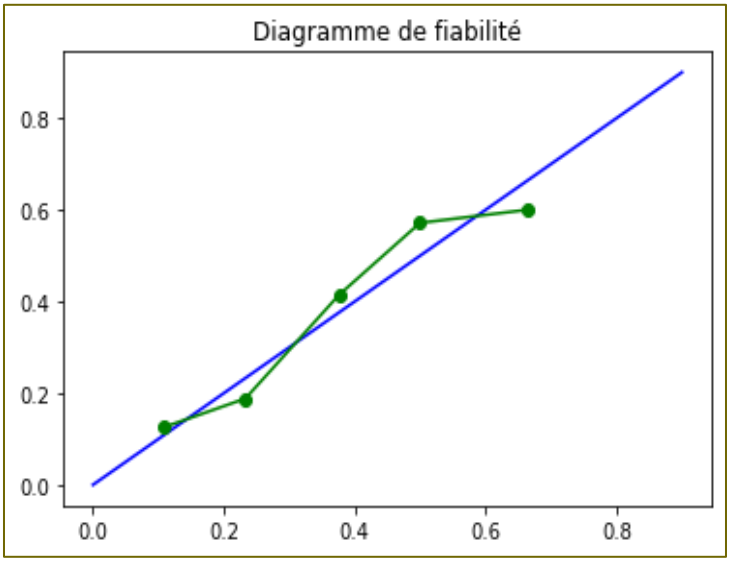

Figure 2 - Diagramme de fiabilité

Pour ce qui est de notre exemple, la modélisation semble relativement intéressante même si nous notons ici ou là des écarts par rapport à la première bissectrice.

3.4.2 Test de Hosmer-Lemeshow

Le test de Hosmer-Lemeshow procède de la même idée de confrontation entre scores estimés et observés. Mais il quantifie les écarts à l'aide d'un indicateur statistique (LIVRE, section 2.3).

```
#data frame temporaire avec y et les scores
df = pandas.DataFrame({"y":yTrain,"score":scores})

#10 intervalles de fréquences égales
intv = pandas.qcut(df.score,q=10)

#intégrées dans le df
df['intv'] = intv
print(df)
```

```
     y     score              intv
0    0  0.098404    (0.0897, 0.13]
1    1  0.590965    (0.487, 0.729]
2    0  0.175729     (0.17, 0.205]
```

```
3    0  0.098360    (0.0897, 0.13]
4    0  0.253698     (0.233, 0.27]
..  ..       ...               ...
396  0  0.218325    (0.205, 0.233]
397  1  0.490415    (0.487, 0.729]
398  0  0.042293  (0.0287, 0.0897]
399  0  0.138713     (0.13, 0.17]
400  0  0.070765  (0.0287, 0.0897]

[401 rows x 3 columns]
```

A partir de ce tableau, nous pouvons retracer les étapes du calcul de l'indicateur statistique (LIVRE, section 2.3.1).

```
#effectifs par groupe
n_tot = df.pivot_table(index="intv",values="y",aggfunc="count").values[:,0]
print(n_tot)
```

```
[41 40 40 40 40 40 40 40 40 40]
```

```
#somme des scores par groupes
s_scores = df.pivot_table(index='intv',values="score",aggfunc="sum").values[:,0]
print(s_scores)
```

```
[ 2.84184359  4.34219237  5.84098854  7.43083651  8.69158827 10.08730971
 11.70113388 14.39094303 17.6729413  23.0002228 ]
```

```
#nombre de positifs par groupe
n_pos = df.pivot_table(index="intv",values="y",aggfunc="sum").values[:,0]
print(n_pos)
```

```
[ 6  1  8  7  9  4 10 17 18 26]
```

```
#nombre de négatifs par groupe
n_neg = n_tot - n_pos
print(n_neg)
```

```
[35 39 32 33 31 36 30 23 22 14]
```

Nous nous sommes basés sur l'équation en 2 parties (LIVRE, équation 2.3).

```
C1 = numpy.sum((n_pos - s_scores)**2/s_scores)
print(C1)
```

```
11.707217210943332
```

```
C2 = numpy.sum((n_neg - (n_tot - s_scores))**2/((n_tot - s_scores)))
print(C2)
```

```
2.860830142686717
```

```
#statistique de Hosmer-Lemeshow
HL = C1 + C2
print(HL)
```

```
14.56804735363005
```

Laquelle suite une loi du KHI-2 à 8 degrés de liberté.

```
#librairie scipy
import scipy

#probabilité critique
pvalue = 1.0 - scipy.stats.chi2.cdf(HL,8)
print(pvalue)
```

```
0.0681091008498439
```

Au risque 5%, nous ne pouvons pas rejeter l'hypothèse de compatibilité de notre modèle avec les données.

Remarque : Dans cette partie aurait dû être intégrée la construction de la courbe ROC et le calcul du critère AUC (LIVRE, section 2.5). Mais nous avons préféré réserver ce thème pour le travail sur l'échantillon test (section **Erreur ! Source du renvoi introuvable.**). En effet, nous avions noté dans un de nos exercices sous R (Régression Logistique, TD 4.b) que nous obtenions des résultats biaisés en resubstitution, trop optimistes par rapport aux réelles performances prédictives du modèle.

3.5 Tests de significativité des coefficients

Cette partie reprend la trame du chapitre 3 du LIVRE. Il s'agit de tester la nullité de tout ou partie des coefficients de la régression. Nous nous appuyons sur les 2 approches qui font référence : le test du rapport de vraisemblance et le test de Wald.

3.5.1 Test de significativité globale de la régression

Il s'agit de vérifier si les coefficients, à l'exception de la constante, sont tous simultanément nuls (LIVRE, section 3.2.4).

3.5.1.1 Via le test du rapport de vraisemblance

Ce test est fondé sur la confrontation des déviances du modèle et du null modèle (modèle trivial composé exclusivement de la constante). On se rappelle que la déviance est égale à `(-2) x log-vraisemblance`.

```
#déviance du modèle
dev_modele = (-2) * res.llf
print("Deviance du modèle : %.4f " % (dev_modele))

#déviance du modèle trivial
dev_null = (-2) * res.llnull
print("Deviance du modèle : %.4f " % (dev_null))

#statistique du rapport de vraisemblance
LR_stat = dev_null - dev_modele
print("Stat. du rapport de vraisemblance : %.4f " % (LR_stat))
```

```
#laquelle était fournie directement par l'objet
print("Stat. du rapport de vraisemblance via l'objet résultat : %.4f" % (res.llr))
```

```
Deviance du modèle : 415.3439
Deviance du modèle : 463.1925
Stat. du rapport de vraisemblance : 47.8486
Stat. du rapport de vraisemblance via l'objet résultat : 47.8486
```

Notre calcul rejoint la valeur renvoyée par la propriété dédiée de l'objet régression. C'est toujours rassurant. Le degré de liberté est égal au nombre de paramètres retirés du modèle. Sous l'hypothèse nulle (tous les coefficients sont simultanément égaux à zéro), la statistique suit une loi du KHI-2.

```
#degré de liberté du test (nb. de coef. estimés excepté la constante)
print(res.df_model)
```

```
8.0
```

```
#p-value du test
pvalue = 1.0 - scipy.stats.chi2.cdf(res.llr,res.df_model)
print(pvalue)
```

```
1.0560156704642054e-07
```

```
#laquelle était également fournie par l'objet
print(res.llr_pvalue)
```

```
1.0560156705523494e-07
```

Encore une fois, nos calculs concordent avec la valeur de la p-value restituée par l'objet. Au risque 5%, nous pouvons rejeter l'hypothèse de nullité des coefficients.

3.5.1.2 Via les comparaisons des critères AIC et BIC

Une autre approche méconnue pour évaluer la qualité globale du modèle consiste à confronter les critères AIC (Akaike ; LIVRE, section 7.2.1, équation 7.2) ou BIC (Schwartz ; LIVRE, section 7.2.1, équation 7.3) du modèle étudié avec ceux du modèle trivial (null modèle).

Si (AIC_modèle < AIC_null), nous pouvons conclure que notre modèle est globalement pertinent (idem pour le critère BIC).

Critère AIC. L'AIC du modèle est restituée par une des propriétés de l'objet. Nous devons en revanche calculer l'AIC_null à partir de la log-vraisemblance du modèle trivial.

```
#AIC du modèle
print("AIC du modèle : %.4f" % (res.aic))

#AIC du modèle trivial - 1 seul param. estimé, la constante
aic_null = (-2) * res.llnull + 2 * (1)
print("AIC du modèle trivial : %.4f" % (aic_null))
```

```
AIC du modèle : 433.3439
AIC du modèle trivial : 465.1925
```

Apparemment, notre modèle est globalement pertinent.

Critère BIC. Nous pouvons adopter la même démarche pour le critère BIC. Ce dernier a la particularité de pénaliser davantage la complexité (le nombre de paramètres à estimer). Il nous indique que le modèle n'est pas si pertinent que cela finalement.

```
#BIC du modèle
print("BIC du modèle : %.4f" % (res.bic))

#BIC du modèle trivial - 1 seul param. estimé, la constante
bic_null = (-2) * res.llnull + numpy.log(n) * (1)
print("BIC du modèle trivial : %.4f" % (bic_null))
```

```
BIC du modèle : 469.2895
BIC du modèle trivial : 469.1865
```

3.5.2 Tester la significativité d'un coefficient

Le test le plus immédiat est celui de Wald (LIVRE, section 3.3.2). Il suffit de regarder les p-value associées à chaque variable dans le résumé des résultats (summary, page 6). Les coefficients non-significativement différents de 0 au risque 5% - dont la p-value est supérieure à 0.05 - sont ceux des variables : Sex, Education, Occupation.

Mais nous pouvons également passer par un test de vraisemblance (LIVRE, section 3.2.2). En confrontant les résultats de la régression avec et sans la variable incriminée. Par exemple, pour évaluer le coefficient associé à la variable "Sex". Nous effectuons la régression en l'excluant.

```
#data frame sans "Sex"
XTrain_wo_Sex = XTrainBis.drop(columns=['Sex'])
print(XTrain_wo_Sex.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 401 entries, 0 to 400
Data columns (total 8 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   const           401 non-null    float64
 1   Age             401 non-null    float64
 2   YearsMarried    401 non-null    float64
 3   Children        401 non-null    int64
 4   Religious       401 non-null    int64
 5   Education       401 non-null    int64
 6   Occupation      401 non-null    int64
 7   RatingMarriage  401 non-null    int64
dtypes: float64(3), int64(5)
```

```
#régression sans Sex
lr_wo_Sex = Logit(yTrain,XTrain_wo_Sex)

#résultats
res_wo_Sex = lr_wo_Sex.fit()
```

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

```
Optimization terminated successfully.
         Current function value: 0.520052
         Iterations 6
                           Logit Regression Results
==============================================================================
Dep. Variable:             Infidelite   No. Observations:                  401
Model:                          Logit   Df Residuals:                      393
Method:                           MLE   Df Model:                            7
Date:                Mon, 30 Mar 2020   Pseudo R-squ.:                0.09955
Time:                        17:59:36   Log-Likelihood:                -208.54
converged:                       True   LL-Null:                       -231.60
Covariance Type:            nonrobust   LLR p-value:                 8.319e-08
=====================================================================================
                    coef    std err          z      P>|z|      [0.025      0.975]
-------------------------------------------------------------------------------------
const              1.1048      1.033      1.070      0.285      -0.919       3.129
Age               -0.0530      0.022     -2.386      0.017      -0.096      -0.009
YearsMarried       0.0910      0.038      2.402      0.016       0.017       0.165
Children           0.8432      0.378      2.231      0.026       0.103       1.584
Religious         -0.2505      0.109     -2.304      0.021      -0.464      -0.037
Education          0.0228      0.059      0.388      0.698      -0.092       0.138
Occupation         0.0684      0.079      0.867      0.386      -0.086       0.223
RatingMarriage    -0.4589      0.108     -4.260      0.000      -0.670      -0.248
=====================================================================================
```

La statistique de test est obtenue par l'écart entre les déviances.

```
#statistique de test - différence entre les déviances
LR_Sex = (-2) * res_wo_Sex.llf - (-2) * res.llf
print(LR_Sex)
```

```
1.7378428085156088
```

Sous H0, le coefficient associé à « Sex » est nul, elle suit une loi du KHI-2 à 1 degré de liberté.

```
#degré de liberté = 1 puisqu'un seul coef. retiré
ddl = res_wo_Sex.df_resid - res.df_resid
print("Degré de liberté du test : %.d" % (ddl))

#p-value
pvalue = 1.0 - scipy.stats.chi2.cdf(LR_Sex,ddl)
print("Probabilité critique : %.4f" % (pvalue))
```

```
Degré de liberté du test : 1
Probabilité critique : 0.1874
```

Très proche du résultat obtenu avec Wald finalement (en termes de p-value tout du moins). Ce n'est pas très étonnant. Notre effectif est assez grand. L'approximation normale est d'autant meilleure que le ratio entre le nombre d'observations et le nombre de variables augmente.

3.5.3 Tester un groupe de coefficients

3.5.3.1 Le plus simple, passer par la vraisemblance

L'approche est aisément généralisable au test de significativité de plusieurs coefficients. Par exemple, vérifions que les coefficients de (Sex, Education et Occupation) peuvent être considérés comme simultanément nuls (H0 : a_Sex = a_Education = a_Occupation = 0).

```
#définir la matrice des X sans les 3 variables
XTrain_wo_3 = XTrainBis.drop(columns=['Sex','Education','Occupation'])
print(XTrain_wo_3.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 401 entries, 0 to 400
Data columns (total 6 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   const           401 non-null    float64
 1   Age             401 non-null    float64
 2   YearsMarried    401 non-null    float64
 3   Children        401 non-null    int64  
 4   Religious       401 non-null    int64  
 5   RatingMarriage  401 non-null    int64  
dtypes: float64(3), int64(3)
memory usage: 18.9 KB
None
```

Nous réalisons la régression sans les 3 variables.

```
#régression sans les 3 variables
lr_wo_3 = Logit(yTrain,XTrain_wo_3)

#résultats
res_wo_3 = lr_wo_3.fit()
```

Nous pouvons par la suite réaliser le test en calculant la statistique du rapport de vraisemblance (par l'écart entre les déviances) et calculer la p-value avec les degrés de liberté idoines.

```
#statistique de test - différence entre les déviances
LR_3 = (-2) * res_wo_3.llf - (-2) * res.llf
print("Statistique de test : %.4f" % (LR_3))

#degré de liberté = 3 puisque 3 coef. retirés
ddl = res_wo_3.df_resid - res.df_resid
print("Degré de liberté du test : %.d" % (ddl))

#p-value
pvalue = 1.0 - scipy.stats.chi2.cdf(LR_3,ddl)
print("Probabilité critique : %.4f" % (pvalue))
```

```
Statistique de test : 3.4886
Degré de liberté du test : 3
Probabilité critique : 0.3222
```

Conclusion : La nullité simultanée des coefficients de (Sex, Education, Occupation) n'est pas démentie par les données au risque 5%.

3.5.3.2 Test de Wald – Calcul matriciel

Au prix d'une gymnastique matricielle un peu complexe, il est également possible de s'appuyer sur la normalité asymptotique des estimateurs du maximum de vraisemblance, et d'appliquer le principe du test de Wald, pour tester la nullité simultanée de plusieurs coefficients (LIVRE, section 3.3.4). Le secret réside dans l'utilisation à bon escient des covariances estimées des coefficients estimés puis les variables, et donc les coefficients, ne sont pas indépendantes.

Nous testons de nouveau la nullité des coefficients de (Sex, Education, Occupation).

```
#afficher la matrice de var-covar des coefs. estimés.
print(res.cov_params())

#récupération sous une forme matricielle
vcov = res.cov_params().values
```

```
                  const       Sex       Age  YearsMarried  Children  \
const          1.131344  0.079738 -0.009454      0.011457 -0.078939
Sex            0.079738  0.086144 -0.001002      0.002000 -0.015955
Age           -0.009454 -0.001002  0.000502     -0.000619 -0.000239
YearsMarried   0.011457  0.002000 -0.000619      0.001469 -0.004560
Children      -0.078939 -0.015955 -0.000239     -0.004560  0.145868
Religious     -0.034708 -0.001810  0.000079     -0.000721 -0.001280
Education     -0.042083 -0.002403 -0.000062      0.000027 -0.000541
Occupation     0.009881 -0.009562 -0.000218      0.000005  0.005387
RatingMarriage -0.036435 -0.002619  0.000205      0.000079  0.001316

                Religious  Education  Occupation  RatingMarriage
const           -0.034708  -0.042083    0.009881       -0.036435
Sex             -0.001810  -0.002403   -0.009562       -0.002619
Age              0.000079  -0.000062   -0.000218        0.000205
YearsMarried    -0.000721   0.000027    0.000005        0.000079
Children        -0.001280  -0.000541    0.005387        0.001316
Religious        0.011954   0.000297    0.000329       -0.000509
Education        0.000297   0.003513   -0.002200       -0.000855
Occupation       0.000329  -0.002200    0.007358        0.000389
RatingMarriage  -0.000509  -0.000855    0.000389        0.011749
```

```
#nous avons bien le carré des ecarts-type estimés sur la diagonale
#à confronter avec les sorties de summary (page 6)
print(numpy.sqrt(numpy.diagonal(vcov)))
```

```
[1.06364633 0.29350264 0.02240929 0.03832698 0.38192712 0.10933331
 0.05926898 0.08578091 0.10839137]
```

Reproduisons les étapes de notre support pour (H0 : a_Sex = a_Education = a_Occupation = 0).

Récupération de la sous-partie de la matrice de variance covariance qui nous concerne.

```
#indice des coefficients concernés
indices = [1,6,7]

#sous-matrice de var.covar
subset_vcov = numpy.zeros(shape=(3,3))
for i in range(3):
    for j in range(3):
        subset_vcov[i,j] = vcov[indices[i],indices[j]]

#vérification
print(subset_vcov)
```

```
[[ 0.0861438  -0.00240294 -0.00956197]
 [-0.00240294  0.00351281 -0.00219997]
 [-0.00956197 -0.00219997  0.00735837]]
```

Nous inversions ensuite cette sous-matrice.

```
#inversion de cette matrice
inv_subset_vcov = numpy.linalg.inv(subset_vcov)
print(inv_subset_vcov)
```

```
[[ 15.61981453  28.78641268  28.90389609]
 [ 28.78641268 403.30529246 157.98548204]
 [ 28.90389609 157.98548204 220.69326558]]
```

Nous récupérons le sous-vecteur des coefficients estimés.

```
#coefficients estimés
a = res.params[indices].values
```

Reste à calculer la forme quadratique correspondant à la statistique de test (LIVRE, équation 3.5) et la probabilité critique associée.

```
#produit matriciel
stat_3 = numpy.dot(a,numpy.dot(inv_subset_vcov,a))
print("Stat. de test : %.4f" % (stat_3))

#p-value (ddl = 3 puisque 3 coef. à tester)
pvalue = 1.0 - scipy.stats.chi2.cdf(stat_3,3)
print("p-value : %.4f" % (pvalue))
```

```
Stat. de test : 3.4552
p-value : 0.3266
```

Encore une fois, du fait de l'effectif assez élevé, les résultats sont très similaires à ceux du test du rapport de vraisemblance (section 3.5.3.1).

3.5.3.3 Test de Wald avec l'outil dédié

En réalité, on s'est un peu embêté pour rien dans la section précédente. « Statsmodels » propose l'outil wald_test() pour réaliser les tests généralisés (LIVRE, section 3.3.6). L'enjeu réside alors dans

la définition de la matrice M permettant de spécifier les coefficients à tester (a_Sex en position 1, a_Education en 6, a_Occupation en 7). Les résultats sont complétement cohérents.

```
#matrice des coefficients à tester
M = [[0,1,0,0,0,0,0,0,0],[0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,1,0]]

#calculer la stat. de test
stat_3bis = res.wald_test(M)
print(stat_3bis)
```

```
<Wald test (chi2): statistic=[[3.45523895]], p-value=0.32661404952574724, df_denom=3>
```

3.6 Performances prédictives

Apprécier le modèle sur un second jeu de données à part, communément appelé « échantillon test », permet d'obtenir une estimation (plus) représentative de ses performances dans la population. On parle de schéma d'évaluation « holdout » (« Validation croisée, Bootstrap – Diapos », février 2015). Nous utilisons les données issues de la seconde feuille « **test** » de notre classeur Excel « infidelites_python.xlsx » dans cette section.

3.6.1 Prédiction et matrice de confusion

Comme pour en apprentissage (section 3.1.1), il faut tout d'abord ajouter la colonne de constante dans la matrice des variables explicatives.

```
#préparation de l'échantillon test
#par adjonction de la constante
XTest_Bis = add_constant(XTest)
print(XTest_Bis.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   const           200 non-null    float64
 1   Sex             200 non-null    int64
 2   Age             200 non-null    float64
 3   YearsMarried    200 non-null    float64
 4   Children        200 non-null    int64
 5   Religious       200 non-null    int64
 6   Education       200 non-null    int64
 7   Occupation      200 non-null    int64
 8   RatingMarriage  200 non-null    int64
dtypes: float64(3), int64(6)
memory usage: 14.2 KB
```

Nous réalisons la prédiction en faisant appel à la fonction predict().

```
#calcul de la prédiction sur l'échantillon test
predProbaSm = res.predict(XTest_Bis)

#à l'évidence nous avons les probabilités d'affectation
print(predProbaSm.describe())
```

```
count    200.000000
mean       0.257187
std        0.157722
min        0.025342
25%        0.130654
50%        0.218863
75%        0.374566
max        0.696005
dtype: float64
```

Nous comprenons à travers les statistiques descriptives, parce que ce comportement peut varier d'un outil à l'autre (cf. par exemple pour « Scikit-Learn » plus bas), que nous obtenons les probabilités d'affectation à la classe cible.

Nous les convertissons en classes prédites {0, 1} en les comparant avec la valeur seuil **0.5**.

```
#convertir en prédiction brute
predSm = numpy.where(predProbaSm > 0.5, 1, 0)
print(numpy.unique(predSm,return_counts=True))
```

```
(array([0, 1]), array([184,  16], dtype=int64))
```

Sur les 200 individus en test, 184 sont associés à la classe « 0 », 16 à « 1 ». Nous formons la matrice de confusion en opposant les classes observées et prédites (LIVRE, section 2.1).

```
#matrice de confusion
mcSm = pandas.crosstab(yTest,predSm)
print(mcSm)
```

```
col_0         0  1
Infidelite
0           147  9
1            37  7
```

```
#transformer en matrice Numpy
mcSmNumpy = mcSm.values
```

Et nous pouvons en déduire les différents indicateurs de performances, notamment le taux de reconnaissance (ou taux de succès) et le taux d'erreur.

```
#taux de reconnaissance
accSm = numpy.sum(numpy.diagonal(mcSmNumpy))/numpy.sum(mcSmNumpy)
print("Taux de reconnaissance : %.4f" % (accSm))

#taux d'erreur
errSm = 1.0 - accSm
print("Taux d'erreur' : %.4f" % (errSm))
```

```
Taux de reconnaissance : 0.7700
Taux d'erreur' : 0.2300
```

3.6.2 Courbe ROC en test

Pour construire la courbe ROC (receiver operating characteristics) sur l'échantillon test, nous avons besoin des valeurs observées de la classe et des probabilités d'affectation fournies par le modèle. Nous pouvons le construire explicitement comme nous l'avions réalisé dans notre TD 4.a. Nous préférons aller à l'essentiel dans ce document. Notre objectif est d'explorer les outils statistiques à notre disposition sous Python. Nous utilisons la fonction roc_curve() du package « scikit-learn », que nous étudierons de manière approfondie plus loin (section 4).

```
#importer le module metrics
#de la librairie scikit-learn
import sklearn.metrics as metrics

#colonnes pour les courbes ROC
#fpr (false positive rate -- taux de faux positifs) en abscisse
#tpr (true positive rate – taux de vrais positifs) en ordonnée
#pos_label = 1 pour indiquer la modalité cible
fprSm, tprSm, _ = metrics.roc_curve(yTest,predProbaSm,pos_label=1)

#graphique -- construire la diagonale de référence
#cas du modèle qui ne fait pas mieux que l'affectation des probabilités
#au hasard – notre courbe ne doit pas passer en dessous
#plus il s'en écarte vers le haut, mieux c'est
plt.plot(numpy.arange(0,1.1,0.1),numpy.arange(0,1.1,0.1),'b')

#rajouter notre diagramme
plt.plot(fprSm,tprSm,"g")

#titre
plt.title("Courbe ROC")

#faire apparaître le graphique
plt.show()
```

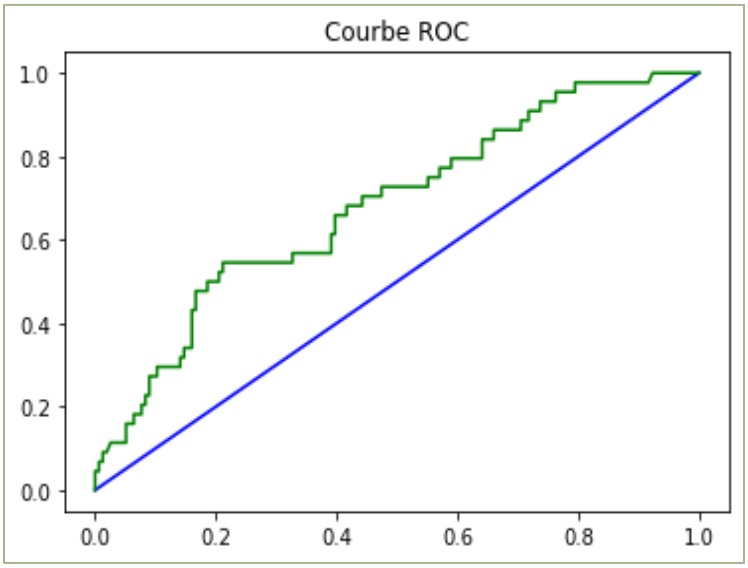

Figure 3 - Courbe ROC sur l'échantillon test

Pour calculer l'AUC (aire sous la courbe), nous faisons appel à la fonction roc_auc_score()

```
#valeur de l'AUC
aucSm = metrics.roc_auc_score(yTest,predProbaSm)
print("AUC : %.4f" % (aucSm))
```

```
AUC : 0.6808
```

4 Régression logistique avec "scikit-learn"

La libraire « scikit-learn » est très populaire auprès des data scientists. Le tutoriel que je lui ai consacré est le plus consulté sur mon blog (« Python – Machine Learning avec Scikit-Learn », septembre 2015). On le comprend aisément à l'usage : elle est très complète ; son mode opératoire est particulièrement cohérent, quelle que soit la famille de méthode de machine learning que nous utilisons ; il est très facile d'enchaîner des opérations complexes, notamment avec le mécanisme des « pipeline ». Voyons comment se comporte l'algorithme de régression logistique qu'elle propose.

4.1 Importation et vérification de version

Sage précaution toujours, nous affichons le numéro de version de la librairie.

```
#importation
import sklearn

#version
print(sklearn.__version__)
```

```
0.22.2.post1
```

4.2 Régression logistique avec scikit-learn

En réalité, plusieurs outils permettent de lancer quelque chose qui ressemble à la régression logistique avec scikit-learn. Nous choisissons la classe LogisticRegression dont la désignation est la plus évidente. Autant « statsmodels » était d'obédience statistique, autant « scikit-learn » est imprégnée de la culture machine learning, tournée essentiellement vers l'efficacité des calculs et la performance prédictive. L'inférence statistique est ignorée ostensiblement.

4.2.1 Régression sur données originelles

Nous instancions une régression logistique sans pénalité c.-à-d. ni Ridge, ni Lasso. Nous lançons l'estimation des paramètres avec la fonction fit() qui prend en entrée les données en apprentissage avec la matrice des explicatives (il n'est pas nécessaire d'adjoindre la constante) et le vecteur des classes observées.

```
#importation de la classe de calcul
from sklearn.linear_model import LogisticRegression

#instanciation
lrSk = LogisticRegression(penalty='none')

#lancement des calculs
#pas nécessaire de rajouter la constante
lrSk.fit(XTrain,yTrain)
```

```
D:\Logiciels\Anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:940:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

Un message sur fond rouge (sous Jupyter Notebook) n'est jamais engageant. L'outil nous annonce que le processus n'a pas convergé alors que, dixit la documentation, le nombre maximal d'itération par défaut est (max_iter = 100). Rappelons que sous « statsmodels », 6 itérations ont été suffisantes (page 5). Pour solutionner le problème, « Scikit-learn » nous propose de : soit augmenter le nombre d'itérations (un peu bourrin quand-même comme solution), soit de standardiser les données avant de lancer l'algorithme (ah bon ? quel rapport avec la régression ? pourquoi nous n'avons pas eu à le faire sous « statsmodels » ?).

Nous reviendrons sur cet écueil dans la section suivante. Pour l'instant, affichons les coefficients estimés, ceux des variables…

```
#affichage des coefficients
print(pandas.DataFrame({"var":XTrain.columns,"coef":lrSk.coef_[0]}))
```

```
              var      coef
0             Sex  0.405682
1             Age -0.054559
2    YearsMarried  0.096893
3        Children  0.729334
4       Religious -0.247918
5       Education  0.015070
6      Occupation  0.019415
7  RatingMarriage -0.451208
```

… et la constante.

```
#la constante
print(lrSk.intercept_)
```

```
[1.29359774]
```

Force est de constater qu'il y a eu un problème durant le processus d'apprentissage. Les coefficients estimés diffèrent sensiblement de ceux obtenus sous « statsmodels » (page 6) et sous « Tanagra » (Figure 1).

4.2.2 Régression logistique sur données standardisées

La documentation de la régression logistique sous « scikit-learn » nous indique les différents algorithmes d'optimisation qu'il est susceptible d'utiliser (option « solver »), qui sont pour la plupart des succédanés de la descente du gradient. Ces approches, certains plus que d'autres, sont sensibles aux différences d'échelles entre les variables, d'où l'indication « scale the data » dans le « warning » envoyé par la méthode fit().

Dans ce qui suit, nous centrons et réduisons les données d'apprentissage avant de relancer la modélisation.

```
#importation de l'outil
from sklearn import preprocessing

#instanciation
stds = preprocessing.StandardScaler()

#transformation
ZTrain = stds.fit_transform(XTrain)
print(scipy.stats.describe(ZTrain,axis=0,ddof=0))
```

```
DescribeResult(nobs=401, minmax=(array([…]), array([…])),
       mean=array([ 9.74559862e-17, -1.86052337e-16, -3.10087229e-17, -1.99341790e-17,
        -1.90482155e-16, -3.16731955e-16,  1.55043614e-16, -1.50613797e-16]),
       variance=array([1., 1., 1., 1., 1., 1., 1., 1.]),
       skewness=array([ …]), kurtosis=array([…]))
```

Les variables sont bien de moyenne nulle et d'écart-type unitaire.

Nous instancions une nouvelle version de la régression et nous affichons les coefficients estimés.

```
#instanciation
lrSkStd = LogisticRegression(penalty='none')

#lancement des calculs -- pas nécessaire de rajouter la constante
lrSkStd.fit(ZTrain,yTrain)

#affichage des coefficients
print(pandas.DataFrame({"var":XTrain.columns,"coef":lrSkStd.coef_[0]}))
```

```
              var      coef
0             Sex  0.192273
1             Age -0.545048
2    YearsMarried  0.560184
3        Children  0.334748
4       Religious -0.298916
5       Education  0.027729
6      Occupation  0.048940
7  RatingMarriage -0.521574
```

Pas de message d'erreur ou de warning intempestifs cette fois-ci. Mais les coefficients obtenus sont très différents de ceux de « statsmodels » (page 6) puisque nous travaillons sur des données transformées. Nous devons les « dé-standardiser » en les divisant par les écarts-type des variables.

```
#correction des coefficients - dé-standardisation
#par les écarts-type utilisés lors de la standardisation des variables
coefUnstd = lrSkStd.coef_[0] / stds.scale_

#affichage des coefficients corrigés
print(pandas.DataFrame({"var":XTrain.columns,"coef":coefUnstd}))
```

```
              var      coef
0             Sex  0.385556
1             Age -0.057157
2    YearsMarried  0.099488
3        Children  0.766189
4       Religious -0.258022
5       Education  0.011254
6      Occupation  0.026307
7  RatingMarriage -0.468419
```

Maintenant seulement les paramètres estimés sont tout à fait cohérents (page 6). Note : Il y a quand-même de petites différences mais elles sont négligeables dans la mesure où une solution exacte n'est pas possible de toute manière. Les heuristiques de calcul engendre une précision approximative que l'on peut piloter avec l'option « tol » (pour tolérance d'erreur).

Nous devons procéder de même avec la constante estimée de la régression mais, en sus des écarts-type, les moyennes des variables entrent également dans le calcul.

```
#pour la constante, l'opération est plus complexe
interceptUnStd = lrSkStd.intercept_ + numpy.sum(lrSkStd.coef_[0]*(-stds.mean_/stds.scale_))
print(interceptUnStd)
```

```
[1.46548767]
```

4.2.3 Calcul de la log-vraisemblance

Les sorties de « scikit-learn » sont peu dissertes. Nous n'avons pas d'indications sur la valeur de la log-vraisemblance à l'issue du processus d'optimisation. Dans cette section, nous la calculons explicitement à partir des classes observées et des probabilité d'appartenance aux classes fournies par la méthode predict_proba().

```
#probabilités d'affectation
proba01 = lrSkStd.predict_proba(ZTrain)

#affichage des 5 premières valeurs
print(proba01[:5,:])
```

```
[[0.90159516 0.09840484]
 [0.40904169 0.59095831]
 [0.82428084 0.17571916]
 [0.90164012 0.09835988]
 [0.74631005 0.25368995]]
```

Les probabilités sont sur deux colonnes. La première, n°0, pour l'appartenance à (Y = 0), la seconde pour (Y = 1). Nous récupérons cette dernière.

```
#récupération de la colonne n°1
proba1 = proba01[:,1]

#description stat
print(scipy.stats.describe(proba1))
```

```
DescribeResult(nobs=401, minmax=(0.029675984415899494, 0.7289302284184851),
mean=0.26434082671891773, variance=0.023199197026083168, skewness=0.8253545254254041,
kurtosis=0.1620167496251228)
```

Nous pouvons calculer la log-vraisemblance (LIVRE, équation 1.8).

```
#log-vraisemblance
log_likelihood = numpy.sum(yTrain*numpy.log(proba1)+(1.0-yTrain)*numpy.log(1.0-proba1))
print(log_likelihood)
```

```
-207.67194817327652
```

Encore une fois, nous sommes cohérents avec « statsmodels » (page 6).

4.3 Performances prédictives en test

Pour appliquer le modèle sur l'échantillon test, il faut au préalable administrer la même transformation (centrage, réduction) sur ce second dataset en utilisant les paramètres (moyennes, écarts-type) calculés sur l'échantillon d'apprentissage.

```
#transformation de l'échantillon test
ZTest = stds.transform(XTest)

#stat. descriptives
print(scipy.stats.describe(ZTest,axis=0,ddof=0))
```

```
DescribeResult(nobs=200, minmax=(),
      mean=array([ 0.07250906, -0.10294845, -0.06287895, -0.19030003, -0.06012224,
        0.01436194,  0.04457129,  0.11201462]),
      variance=array([1.00525756, 0.83439542, 0.92942511, 1.17559603, 1.03949527,
       0.84731025, 0.86322436, 0.9313974 ]),
      skewness=array(), kurtosis=array())
```

Les moyennes et écarts-type calculés ne sont pas forcément nuls et unitaires sur ce second dataset transformé puisque les moyennes et écarts-type utilisées proviennent d'un autre échantillon (d'apprentissage).

Nous appliquons le modèle (predict) et nous calculons la matrice de confusion avec la fonction confusion_matrix() du module « metrics » de « scikit-learn ».

```
#appliquer la prédiction
predSk = lrSkStd.predict(ZTest)
print(numpy.unique(predSk,return_counts=True))
```

```
(array([0, 1], dtype=int64), array([184,  16], dtype=int64))
```

```
#matrice de confusion
print(metrics.confusion_matrix(yTest,predSk))
```

```
[[147   9]
 [ 37   7]]
```

Nous pouvons calculer individuellement les différents indicateurs mais, avec la fonction classification_report(), nous disposons directement d' un rapport détaillé.

```
#rapport sur la qualité de prédiction
print(metrics.classification_report(yTest,predSk))
```

```
              precision    recall  f1-score   support

           0       0.80      0.94      0.86       156
           1       0.44      0.16      0.23        44

    accuracy                           0.77       200
```

```
   macro avg       0.62      0.55      0.55       200
weighted avg       0.72      0.77      0.73       200
```

La matrice de confusion et, par conséquent, le taux de reconnaissance en test de 77%, sont identiques à ceux de « statsmodels » (section 3.6.1).

5 Conclusion

Nous constatons durant ce tutoriel qu'il est tout à fait possible de mener une analyse probante de régression logistique sous Python. Les deux librairies étudiées, « statsmodels » et « scikit-learn », sont différentes dans leur philosophie et les fonctionnalités proposées. La première, comme son nom l'indique, est empreinte d'une forte culture statistique. La seconde a une approche plutôt « machine learning ». Mais elles se rejoignent dans les résultats. C'est ce qui importe en définitive.

6 Références

[LIVRE] R. Rakotomalala, « Pratique de la régression logistique », version 2.0, mai 2017.

Tutoriel Tanagra, « Python – Econométrie avec StatsModels », septembre 2015.

Scikit-Learn, https://scikit-learn.org/stable/index.html

Statsmodels, https://www.statsmodels.org/stable/index.html