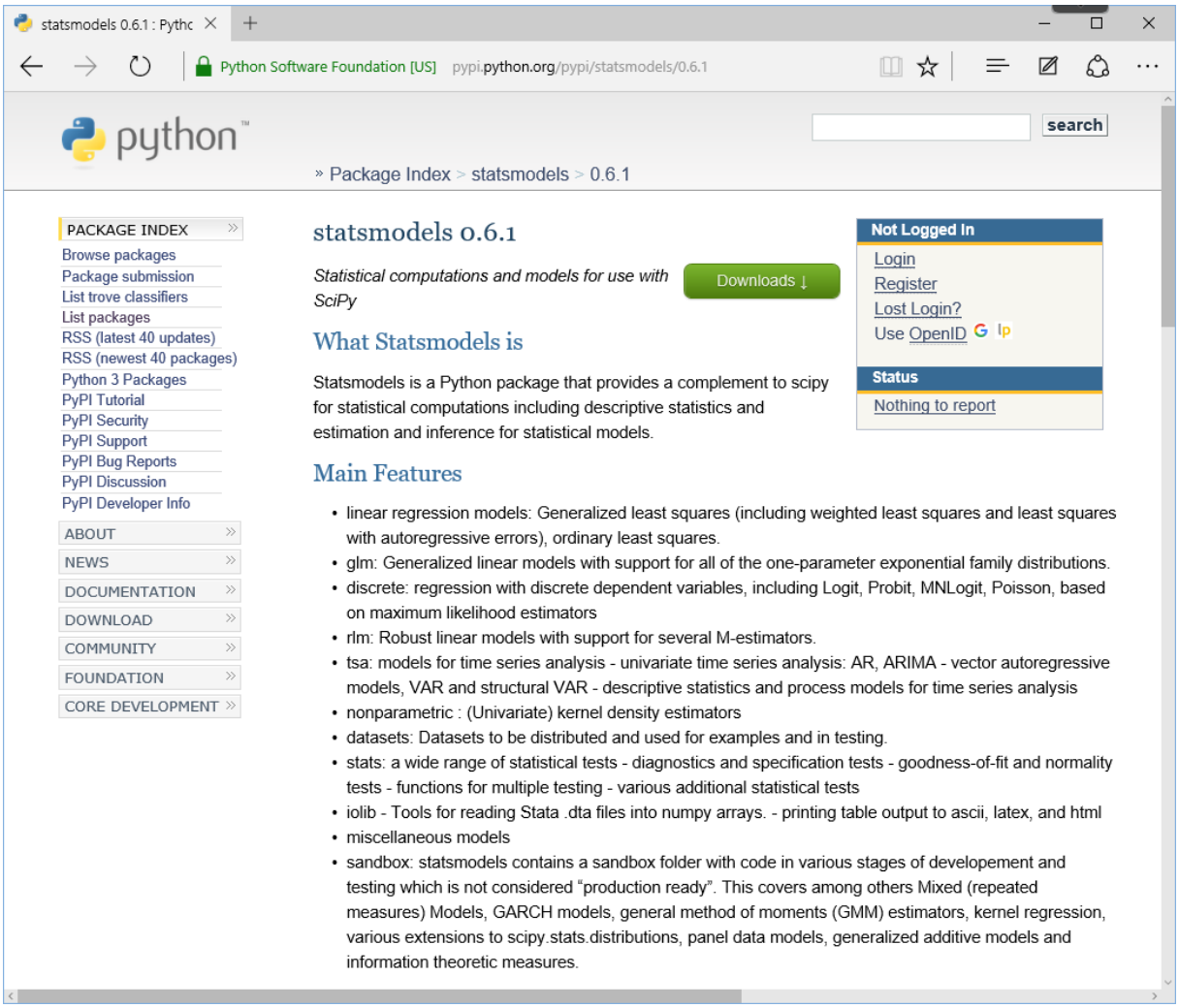
1 Objectif

Présentation du package StatsModels pour Python à travers une étude de cas en économétrie.

StatsModels est un package dédié à la modélisation statistique. Il incorpore un grand nombre de techniques économétriques telles que la régression linéaire, le modèle linéaire généralisé, le traitement des séries temporelles (AR, ARIMA, VAR, etc.). La description de la librairie est accessible sur la page PyPI, le dépôt qui recense les outils et packages consacrés au langage Python.



Dans ce tutoriel, nous essaierons de cerner les potentialités de StatsModels en déroulant une étude de cas en régression linéaire multiple. Nous aborderons tour à tour : l'estimation des paramètres du modèle à l'aide de la méthode des moindres carrés ordinaires, la mise en œuvre de quelques tests statistiques, la vérification de la compatibilité de la distribution des

résidus avec l'hypothèse de normalité, la détection des points atypiques et influents, l'analyse de la colinéarité, la prédiction ponctuelle et par intervalle.

Pour le lecteur désireux de faire le lien entre les commandes utilisées dans Python et les formules économétriques sous-jacentes, je mettrai en référence deux des ouvrages que j'utilise pour mon cours d'économétrie à l'Université : **[Régression]**, « Econométrie - La régression linéaire simple et multiple », mai 2015 ; et **[Diagnostic]**, « Pratique de la Régression Linéaire Multiple - Diagnostic et sélection de variables », mai 2015.

2 Données de modélisation

Nous traitons le fichier « vehicules_1.txt » dans la partie modélisation. Il décrit **n = 26** automobiles à partir de leur cylindrée, leur puissance, leur poids et leur consommation.

modele	cylindree	puissance	poids	conso
Maserati Ghibli GT	2789	209	1485	14.5
Daihatsu Cuore	846	32	650	5.7
Toyota Corolla	1331	55	1010	7.1
Fort Escort 1.4i PT	1390	54	1110	8.6
Mazda Hachtback V	2497	122	1330	10.8
Volvo 960 Kombi aut	2473	125	1570	12.7
Toyota Previa salon	2438	97	1800	12.8
Renault Safrane 2.2. V	2165	101	1500	11.7
Honda Civic Joker 1.4	1396	66	1140	7.7
VW Golt 2.0 GTI	1984	85	1155	9.5
Suzuki Swift 1.0 GLS	993	39	790	5.8
Lancia K 3.0 LS	2958	150	1550	11.9
Mercedes S 600	5987	300	2250	18.7
Volvo 850 2.5	2435	106	1370	10.8
VW Polo 1.4 60	1390	44	955	6.5
Hyundai Sonata 3000	2972	107	1400	11.7
Opel Corsa 1.2i Eco	1195	33	895	6.8
Opel Astra 1.6i 16V	1597	74	1080	7.4
Peugeot 306 XS 108	1761	74	1100	9
Mitsubishi Galant	1998	66	1300	7.6
Citroen ZX Volcane	1998	89	1140	8.8
Peugeot 806 2.0	1998	89	1560	10.8
Fiat Panda Mambo L	899	29	730	6.1
Seat Alhambra 2.0	1984	85	1635	11.6
Ford Fiesta 1.2 Zetec	1242	55	940	6.6

Nous souhaitons expliquer et prédire la consommation des automobiles (y : CONSO) à partir de **p = 3** caractéristiques (X_1 : cylindrée, X_2 : puissance, X_3 : poids).

L'équation de régression s'écrit (Régression, page 87).

$$y_i = a_0 + a_1 x_{i1} + a_2 x_{i2} + a_3 x_{i3} + \varepsilon_i,\ i = 1, \ldots, n$$

L'idée est d'exploiter les données pour estimer au mieux - au sens des moindres carrés ordinaires (MCO) - le vecteur de paramètres (a_0, a_1, a_2, a_3).

3 Chargement des données

Nous utilisons le package Pandas pour le chargement des données. Il propose des fonctionnalités de manipulation de données proches de celles du logiciel R, avec une structure data frame, elle aussi très fortement inspirée de R.

```
#modification du répertoire par défaut
import os
os.chdir("… dossier contenant le fichier de données …")

#utilisation de la librairie Pandas
#spécialisée - entres autres - dans la manipulation des données
import pandas
cars = pandas.read_table("vehicules_1.txt",sep="\t",header=0,index_col=0)
```

Dans l'instruction **read_table()**, nous indiquons : la première ligne correspond aux noms des variables (**header = 0**), la première colonne (n°0) correspond aux identifiants des observations (**index_col = 0**).

Nous affichons les dimensions de l'ensemble de données. Nous les récupérons dans des variables intermédiaires **n** et **p**.

```
#dimensions
print(cars.shape)

#nombre d'observations
n = cars.shape[0]

#nombre de variables explicatives
p = cars.shape[1] - 1
```

Nous avons **n = 26** lignes et 4 colonnes, dont **p = 3** variables explicatives.

Nous affichons le type des colonnes :

```
#liste des colonnes et leurs types
print(cars.dtypes)
```

Les 4 sont numériques (valeurs entières ou réelles). La première (n°0), correspondant aux identifiants des modèles, n'est pas comptabilisée ici.

```
cylindree      int64
```

```
puissance      int64
poids          int64
conso        float64
dtype: object
```

Nous accédons aux identifiants avec la propriété **index**.

```
#liste des modèles)
print(cars.index)
```

Nous obtenons :

```
Index(['Maserati Ghibli GT', 'Daihatsu Cuore', 'Toyota Corolla',
       'Fort Escort 1.4i PT', 'Mazda Hachtback V', 'Volvo 960 Kombi aut',
       'Toyota Previa salon', 'Renault Safrane 2.2. V',
       'Honda Civic Joker 1.4', 'VW Golt 2.0 GTI', 'Suzuki Swift 1.0 GLS',
       'Lancia K 3.0 LS', 'Mercedes S 600', 'Volvo 850 2.5', 'VW Polo 1.4 60',
       'Hyundai Sonata 3000', 'Opel Corsa 1.2i Eco', 'Opel Astra 1.6i 16V',
       'Peugeot 306 XS 108', 'Mitsubishi Galant', 'Citroen ZX Volcane',
       'Peugeot 806 2.0', 'Fiat Panda Mambo L', 'Seat Alhambra 2.0',
       'Ford Fiesta 1.2 Zetec'],
      dtype='object', name='modele')
```

4 Régression

4.1 Lancement des calculs

Nous pouvons passer à l'étape de modélisation en important le package **statsmodels**. Deux options s'offrent à nous. **(1)** La première consiste scinder les données en deux parties : un vecteur contenant les valeurs de la variable cible (exogène) CONSO, une matrice avec les variables explicatives (exogènes) CYLINDREE, PUISSANCE, CONSO. Puis, nous les passons à l'instance de la classe OLS. Cela implique quelques manipulations de données, et surtout de convertir les structures en vecteur et matrice de type numpy. **(2)** La seconde s'appuie sur une classe spécifique (ols) qui reconnaît directement les formules de type R [ex. la fonction lm() de R][1]. La structure data frame de Pandas peut être directement utilisée dans ce cas. Nous optons pour cette solution.

```
#régression avec formule
import statsmodels.formula.api as smf

#instanciation
reg = smf.ols('conso ~ cylindree + puissance + poids', data = cars)
```

[1] Il est également possible de générer les vecteurs et matrices adéquates à partir d'une formule R en utilisant le module patsy.

```
#membres de reg
print(dir(reg))
```

reg est une instance de la classe **ols**. Nous avons accès à ses membres avec l'instruction **dir()** c.-à-d. ses propriétés et méthodes.

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', '_data_attr', '_df_model', '_df_resid',
'_get_init_kwds', '_handle_data', '_init_keys', 'data', 'df_model', 'df_resid',
'endog', 'endog_names', 'exog', 'exog_names', 'fit', 'fit_regularized', 'formula',
'from_formula', 'hessian', 'information', 'initialize', 'k_constant', 'loglike',
'nobs', 'predict', 'rank', 'score', 'weights', 'wendog', 'wexog', 'whiten']
```

Nous utilisons la fonction **fit()** pour lancer le processus de modélisation à partir des données.

```
#lancement des calculs
res = reg.fit()

#liste des membres
print(dir(res))
```

L'objet **res** qui en résulte propose également un grand nombre de propriétés et méthodes.

```
['HC0_se', 'HC1_se', 'HC2_se', 'HC3_se', '_HCCM', '__class__', '__delattr__',
'__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'_cache', '_data_attr', '_get_robustcov_results', '_is_nested',
'_wexog_singular_values', 'aic', 'bic', 'bse', 'centered_tss', 'compare_f_test',
'compare_lm_test', 'compare_lr_test', 'condition_number', 'conf_int',
'conf_int_el', 'cov_HC0', 'cov_HC1', 'cov_HC2', 'cov_HC3', 'cov_kwds',
'cov_params', 'cov_type', 'df_model', 'df_resid', 'eigenvals', 'el_test', 'ess',
'f_pvalue', 'f_test', 'fittedvalues', 'fvalue', 'get_influence',
'get_robustcov_results', 'initialize', 'k_constant', 'llf', 'load', 'model',
'mse_model', 'mse_resid', 'mse_total', 'nobs', 'normalized_cov_params',
'outlier_test', 'params', 'predict', 'pvalues', 'remove_data', 'resid',
'resid_pearson', 'rsquared', 'rsquared_adj', 'save', 'scale', 'ssr', 'summary',
'summary2', 't_test', 'tvalues', 'uncentered_tss', 'use_t', 'wald_test', 'wresid']
```

summary() affiche le détail des résultats.

```
#résultats détaillés
print(res.summary())
```

La présentation est conventionnelle, conforme à ce que l'on observe dans la majorité des logiciels d'économétrie.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   conso   R-squared:                       0.957
Model:                             OLS   Adj. R-squared:                  0.951
Method:                  Least Squares   F-statistic:                     154.7
Date:                 Sun, 27 Sep 2015   Prob (F-statistic):           1.79e-14
Time:                         09:35:48   Log-Likelihood:                -24.400
No. Observations:                   25   AIC:                             56.80
Df Residuals:                       21   BIC:                             61.68
Df Model:                            3
Covariance Type:             nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
Intercept      1.2998      0.614      2.117      0.046         0.023     2.577
cylindree     -0.0005      0.000     -0.953      0.351        -0.002     0.001
puissance      0.0303      0.007      4.052      0.001         0.015     0.046
poids          0.0052      0.001      6.447      0.000         0.004     0.007
==============================================================================
Omnibus:                        0.799   Durbin-Watson:                   2.419
Prob(Omnibus):                  0.671   Jarque-Bera (JB):                0.772
Skew:                          -0.369   Prob(JB):                        0.680
Kurtosis:                       2.558   Cond. No.                     1.14e+04
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.14e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Le modèle, avec un $R^2 = 0.975$, est globalement significatif à 5% [F-statistic = 154.7, avec Prob(F-statistic) = 1.79e-14]. Toutes les variables sont également pertinentes à 5% (cf. t et P>|t| dans le tableau des coefficients) (Régression, chapitre 10, à partir de la page 99). Tout semble aller pour le mieux.

Un avertissement (warnings) nous alerte néanmoins. On suspecte un problème de colinéarité entre les exogènes. Nous le vérifierons plus loin.

4.2 Calculs intermédiaires

L'objet **res** permet de réaliser des calculs intermédiaires. Certains sont relativement simples. Dans le code ci-dessous, nous faisons afficher quelques résultats importants (paramètres estimés, R2) et nous nous amusons à recalculer manuellement la statistique F de significativité globale. Elle coïncide avec celle directement fournie par **res** bien évidemment.

```
#paramètres estimés
print(res.params)

#le R2
print(res.rsquared)

#calcul manuel du F à partir des carrés moyens
F = res.mse_model / res.mse_resid
print(F)

#F fourni par l'objet res
print(res.fvalue)
```

D'autres calculs sont possibles, nous donnant accès à des procédures sophistiquées telles que les tests de contraintes linéaires sur les coefficients (Régression, section 11.3, page 117 ; ce test généralise une grande partie des tests sur les coefficients : significativité, conformité à un standard, comparaisons).

Mettons que nous souhaitons vérifier la nullité de tous les coefficients à l'exception de la constante. Il s'agit ni plus ni moins du test de significativité globale. L'hypothèse nulle peut s'écrire sous une forme matricielle[2] :

$$H_0 : Ra = 0$$

Où

- a = (a_0,a_1,a_2,a_3) est le vecteur des coefficients ;
- R est la matrice de contraintes :

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

[2] statsmodels s'appuie sur une formulation simplifiée où le vecteur de référence **r** (Régression, équation 11.5, page 117) est toujours le vecteur nul c.-à-d. **r = 0**.

En développant un peu, nous avons bien :

$$H_0 : \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\Leftrightarrow H_0 : \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Sous Python, nous décrivons la matrice R puis nous appelons la méthode **f_test()**

```
#test avec combinaison linéaires de param.
#nullité simultanée de tous les param. sauf la constante
import numpy as np

#matrices des combinaisons à tester
R = np.array([[0,1,0,0],[0,0,1,0],[0,0,0,1]])

#on obtient le test F
print(res.f_test(R))
```

Nous retrouvons la statistique F de significativité globale.

```
<F test: F=array([[ 154.67525358]]), p=1.7868899968536844e-14, df_denom=21, df_num=3>
```

5 Diagnostic de la régression

5.1 Normalité des résidus

L'inférence dans la régression linéaire multiple (MCO) repose sur l'hypothèse de normalité des erreurs. Une première vérification importante consiste à vérifier la compatibilité des résidus (l'erreur observée sur l'échantillon) avec ce présupposé.

Test de Jarque-Bera. Nous utilisons le module **stattools** de **statsmodels** pour réaliser le test de Jarque-Bera, basé sur les indicateurs d'asymétrie (Skewness) et d'aplatissement (Kurtosis) de la distribution empirique (Diagnostic, section 1.3.3, page 22).

```
#test de normalité de Jarque-Bera
import statsmodels.api as sm
JB, JBpv,skw,kurt = sm.stats.stattools.jarque_bera(res.resid)
print(JB,JBpv,skw,kurt)
```

Nous avons respectivement : la statistique de test JB, la probabilité critique associée (p-value) JBpv, le coefficient d'asymétrie skw et d'aplatissement kurt.

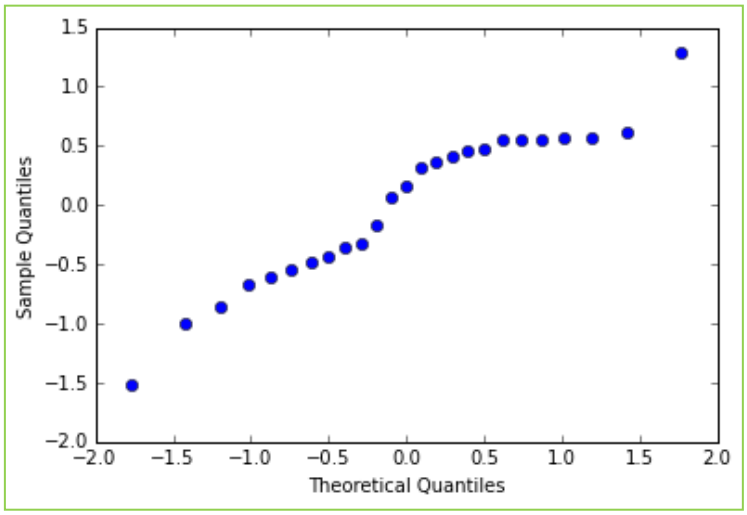
```
0.7721503004927122 0.679719442677 -0.3693040742424057 2.5575948785729956
```

On vérifie très facilement que les indicateurs obtenus (JB, JBpv) sont cohérents avec ceux affichés par l'objet régression dans son résumé (summary) ci-dessus (page 6). L'hypothèse de normalité des erreurs est compatible avec les résidus observés au risque 5%.

Graphique qqplot. Le graphique qqplot permet de vérifier la compatibilité de deux distributions. Dans notre cas, nous confrontons la distribution empirique des résidus avec la distribution théorique de la loi normale. On parle aussi de Droite de Henry dans la littérature. L'hypothèse nulle (normalité) est rejetée si la série de points s'écarte manifestement de la diagonale. Nous utilisons la procédure **qqplot**().

```
#qqpolot vs. loi normale
sm.qqplot(res.resid)
```

Nous avons droit à une sortie graphique.



Le qqplot() confirme à peu près le test de Jarque-Bera, les points sont plus ou moins alignés. Mais il semble y avoir des problèmes pour les grandes valeurs des résidus. Cela laisse imaginer l'existence de points atypiques dans nos données.

5.2 Détection des points atypiques et influents

Nous utilisons un objet dédié, issu du résultat de la régression, pour analyser les points influents.

```
#objet pour les mesures d'influences
infl = res.get_influence()

#membres
print(dir(infl))
```

Il intègre un certain nombre de propriétés et méthodes.

```
['__class__',   '__delattr__',   '__dict__',   '__dir__',   '__doc__',   '__eq__',
'__format__',  '__ge__',  '__getattribute__',  '__gt__',  '__hash__',  '__init__',
'__le__',    '__lt__',    '__module__',    '__ne__',    '__new__',    '__reduce__',
'__reduce_ex__',     '__repr__',     '__setattr__',     '__sizeof__',     '__str__',
'__subclasshook__',  '__weakref__',  '_get_drop_vari',  '_ols_xnoti',  '_res_looo',
'aux_regression_endog',   'aux_regression_exog',   'cooks_distance',   'cov_ratio',
'det_cov_params_not_obsi',   'dfbetas',   'dffits',   'dffits_internal',   'endog',
'ess_press',    'exog',    'get_resid_studentized_external',    'hat_diag_factor',
'hat_matrix_diag', 'influence', 'k_vars', 'model_class', 'nobs', 'params_not_obsi',
'resid_press',              'resid_std',              'resid_studentized_external',
'resid_studentized_internal',    'resid_var',    'results',    'sigma2_not_obsi',
'sigma_est', 'summary_frame', 'summary_table']
```

Levier et résidu standardisé. Nous affichons par exemple les leviers et les résidus standardisés

```
#leviers
print(infl.hat_matrix_diag)

#résidus standardisés
print(infl.resid_studentized_internal)
```

soit, respectivement : leviers,

```
[ 0.72183931  0.17493953  0.06052439  0.05984303  0.05833457  0.09834565
  0.3069384   0.09434881  0.07277269  0.05243782  0.11234304  0.08071319
  0.72325833  0.05315383  0.08893405  0.25018637  0.10112179  0.0540076
  0.05121573  0.10772156  0.05567033  0.16884879  0.13204031  0.24442832
  0.07603257]
```

et résidus standardisés (résidus studentisés internes dans certains ouvrages)

```
[ 1.27808674  0.70998577 -0.71688165  0.81147383  0.10804304  0.93643923
  0.61326258  0.85119912 -1.28230456  0.82173104 -0.4825047  -0.89285915
 -1.47915409  0.46985347 -0.65771702  2.12388261  0.62133156 -1.46735362
  0.84071707 -2.28450314 -0.25846374 -0.56426915  0.84517318  0.26915841
 -0.98733251]
```

L'intérêt de savoir programmer est que nous pouvons vérifier par le calcul les résultats proposés par les différentes procédures. Prenons le résidu standardisé (t_i), nous pouvons l'obtenir à partir de résidu observé $\hat{\varepsilon}_i = y_i - \hat{y}_i$, du levier h_i et de la variance estimée de l'erreur de régression $\hat{\sigma}_\varepsilon^2$ obtenue avec la propriété **scale** de l'objet résultat de régression (**res.scale**) (Diagnostic, équation 2.4, page 34).

$$t_i = \frac{\hat{\varepsilon}_i}{\hat{\sigma}_\varepsilon \sqrt{1-h_i}}$$

Nous écrivons les instructions suivantes :

```
#vérifions avec la formule du cours
import numpy as np
residus = res.resid.as_matrix() #résidus observés
leviers = infl.hat_matrix_diag #levier
sigma_err = np.sqrt(res.scale) #ec.type estimé de l'erreur
res_stds = residus/(sigma_err*np.sqrt(1.0-leviers))
print(res_stds)
```

Les valeurs coïncident exactement avec celles fournies directement la propriété appropriée (resid_studenized_internal). Heureusement, on aurait été bien ennuyé dans le cas contraire.

Résidus studentisés. Nous réitérons l'opération pour le résidu studentisé (résidu studentisé externe dans certains logiciels) (Diagnostic, équation 2.6, page 38)

$$t_i^* = t_i \sqrt{\frac{n-p-2}{n-p-1-t_i^2}}$$

Nous confrontons la propriété resid_studentized_external et le fruit de la formule qui tient en une seule ligne de code.

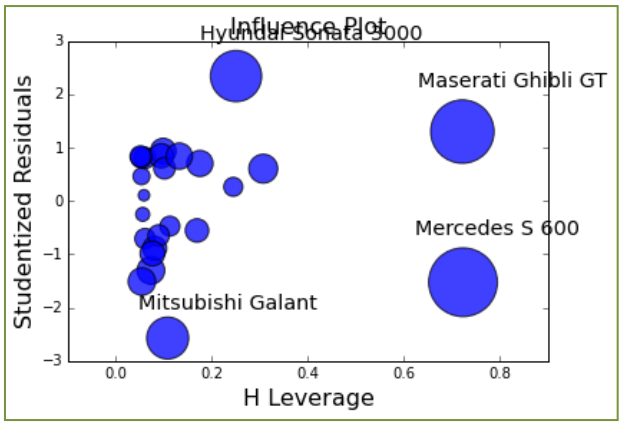
```
#résidus studentisés
print(infl.resid_studentized_external)

#vérifions avec la formule
res_studs = res_stds*np.sqrt((n-p-2)/(n-p-1-res_stds**2))
print(res_studs)
```

Ici également, les valeurs sont cohérentes.

```
[ 1.29882255  0.70134375 -0.70832574  0.80463315  0.10546853  0.933571
  0.60391521  0.8453972  -1.30347228  0.81513961 -0.4735084  -0.88836644
 -1.52514011  0.46095935 -0.64858111  2.33908893  0.61200904 -1.51157649
  0.83462166 -2.57180996 -0.25263694 -0.55489299  0.83920081  0.26312597
 -0.98671171]
```

Graphique. StatsModels propose un outil graphique pour afficher les observations à problème.

```
#graphique des influences()
sm.graphics.influence_plot(res)
```



Les problèmes concernent la Mitsubishi Galant, la Hyundai Sonata, la Mercedes S 600 et la Maserati Ghibli GT.

Détection automatique - Comparaison à un seuil. Inspecter visuellement les données est une bonne chose. Mais pouvoir détecter automatiquement les points atypiques ou influents est préférable, surtout lorsque le nombre d'observations est élevé. Pour ce faire, nous devons définir des valeurs seuils à partir desquelles un point devient suspect.

Levier. Pour le levier, il s'écrit (Diagnostic, équation 2.2, page 32) :

$$s_h = 2 \times \frac{p+1}{n}$$

Un point est potentiellement à problème dès lors que $h_i > s_h$

```
#seuil levier
seuil_levier = 2*(p+1)/n
print(seuil_levier)

#identification
atyp_levier = leviers > seuil_levier
print(atyp_levier)
```

Python nous affiche un vecteur de booléens, les **True** correspondent aux observations atypiques.

```
[ True False False False False False False False False False False False
  True False False False False False False False False False False False]
```

La lecture n'est pas très aisée. Il est plus commode de faire afficher les identifiants des véhicules incriminés.

```
#quels véhicules ?
print(cars.index[atyp_levier],leviers[atyp_levier])
```

Nous avons à la fois les modèles et les valeurs du levier.

```
Index(['Maserati Ghibli GT', 'Mercedes S 600'], dtype='object', name='modele') [
0.72183931  0.72325833]
```

Résidus studentisés. Le seuil est défini par le quantile de la loi de Student à (n-p-2) degrés de liberté (Diagnostic, page 39)

$$s_t = t_{1-0.05/2}(n-p-2)$$

Où t est le quantile d'ordre *0.975* de la loi de Student.

Une observation est suspecte lorsque

$$\left|t_i^*\right| > s_t$$

```
#seuil résidus studentisés
import scipy
seuil_stud = scipy.stats.t.ppf(0.975,df=n-p-2)
print(seuil_stud)

#détection - val. abs > seuil
atyp_stud = np.abs(res_studs) > seuil_stud

#lequels ?
print(cars.index[atyp_stud],res_studs[atyp_stud])
```

Soit,

```
Index(['Hyundai Sonata 3000', 'Mitsubishi Galant'], dtype='object', name='modele')
[ 2.33908893 -2.57180996]
```

Combinaison des critères levier et résidu studentisé. En combinant les deux approches avec l'opérateur logique OR

```
#problématique avec un des deux critères
pbm_infl = np.logical_or(atyp_levier,atyp_stud)
print(cars.index[pbm_infl]))
```

Nous retrouvons les 4 véhicules mis en évidence dans l'outil graphique.

```
Index(['Maserati Ghibli GT', 'Mercedes S 600', 'Hyundai Sonata 3000',
       'Mitsubishi Galant'], dtype='object', name='modele')
```

Autres critères. L'objet propose d'autres critères tels que les DFFITS, les distance de Cook ou encore les DFBETAS (Régression, section 2.5, page 41 et suivantes).

Nous pouvons les présenter sous forme tabulaire.

```
#présentation sous forme de tableau
print(infl.summary_frame().filter(["hat_diag","student_resid","dffits","cooks_d"]))
```

Il faudrait comparer les valeurs avec leurs seuils respectifs :

- $|DFFITS_i| > 2\sqrt{\frac{p+1}{n}}$ pour les DFFITS ;
- $D_i > \frac{4}{n-p-1}$ pour la distance de Cook.

```
                        hat_diag  student_resid    dffits   cooks_d
modele
Maserati Ghibli GT      0.721839       1.298823  2.092292  1.059755
Daihatsu Cuore          0.174940       0.701344  0.322948  0.026720
Toyota Corolla          0.060524      -0.708326 -0.179786  0.008277
Fort Escort 1.4i PT     0.059843       0.804633  0.203004  0.010479
Mazda Hachtback V       0.058335       0.105469  0.026251  0.000181
Volvo 960 Kombi aut     0.098346       0.933571  0.308322  0.023912
Toyota Previa salon     0.306938       0.603915  0.401898  0.041640
Renault Safrane 2.2. V  0.094349       0.845397  0.272865  0.018870
Honda Civic Joker 1.4   0.072773      -1.303472 -0.365168  0.032263
VW Golt 2.0 GTI         0.052438       0.815140  0.191757  0.009342
Suzuki Swift 1.0 GLS    0.112343      -0.473508 -0.168453  0.007366
Lancia K 3.0 LS         0.080713      -0.888366 -0.263232  0.017498
Mercedes S 600          0.723258      -1.525140 -2.465581  1.429505
Volvo 850 2.5           0.053154       0.460959  0.109217  0.003098
VW Polo 1.4 60          0.088934      -0.648581 -0.202639  0.010557
Hyundai Sonata 3000     0.250186       2.339089  1.351145  0.376280
Opel Corsa 1.2i Eco     0.101122       0.612009  0.205272  0.010858
Opel Astra 1.6i 16V     0.054008      -1.511576 -0.361172  0.030731
Peugeot 306 XS 108      0.051216       0.834622  0.193913  0.009538
Mitsubishi Galant       0.107722      -2.571810 -0.893593  0.157516
Citroen ZX Volcane      0.055670      -0.252637 -0.061340  0.000985
Peugeot 806 2.0         0.168849      -0.554893 -0.250103  0.016171
Fiat Panda Mambo L      0.132040       0.839201  0.327318  0.027167
Seat Alhambra 2.0       0.244428       0.263126  0.149659  0.005859
Ford Fiesta 1.2 Zetec   0.076033      -0.986712 -0.283049  0.020054
```

Remarque : Détecter les points atypiques ou influents est une chose, les traiter en est une autre. En effet, on ne peut pas se contenter de les supprimer systématiquement. Il faut identifier pourquoi une observation pose problème, puis déterminer la solution la plus adéquate, qui peut être éventuellement la suppression, mais pas systématiquement. En effet, prenons une configuration simple. Un point peut être atypique parce qu'il prend une valeur inhabituelle sur une variable. Si le processus de sélection des exogènes entraîne son

exclusion du modèle, que faut-il faire alors ? Réintroduire le point ? Laisser tel quel ? Il n'y a pas de solution préétablie. Le processus de modélisation est par nature exploratoire.

6 Détection de la colinéarité

La corrélation entre les exogènes, et de manière plus générale la colinéarité (il existe combinaison linéaire de variables qui est - presque - égale à une constante), fausse l'inférence statistique, notamment parce qu'elle gonfle la variance estimée des coefficients. Il y a différentes manières de l'identifier et de la traiter. Nous nous contentons de quelques techniques de détection basées sur l'analyse de la matrice des corrélations dans cette partie.

Matrice des corrélations. Une règle simple de vérification est de calculer les corrélations croisées entre exogènes et de comparer leur valeur absolue avec 0.8 (« Colinéarité et sélection de variables », page 5). Sous Python, nous isolons les exogènes dans une matrice au format numpy. Nous utilisons par la suite la **corrcoef()** de la librairie scipy.

```
#matrice des corrélations avec scipy
import scipy
mc = scipy.corrcoef(cars_exog,rowvar=0)
print(mc)
```

Il y a manifestement des problèmes dans nos données.

```
[[ 1.          0.94703153  0.87481759]
 [ 0.94703153  1.          0.82796562]
 [ 0.87481759  0.82796562  1.        ]]
```

La cylindrée et la puissance du moteur sont liés, on comprend très bien pourquoi. Elles sont également liées avec le poids des véhicules. Oui, les véhicules lourds ont généralement besoin de plus de puissance.

Règle de Klein. Elle consiste à comparer le carré des corrélations croisées avec le R^2 de la régression. Elle est plus intéressante car elle tient compte des caractéristiques numériques du modèle. Dans notre cas, $R^2 = 0.957$.

```
#règle de Klein
mc2 = mc**2
print(mc2)
```

Aucune des valeurs n'excède le R^2, mais il y a des similarités inquiétantes néanmoins.

```
[[ 1.          0.89686872  0.76530582]
 [ 0.89686872  1.          0.68552706]
 [ 0.76530582  0.68552706  1.        ]]
```

Facteur d'inflation de la variance (VIF). Ce critère permet d'évaluer la liaison d'une exogène avec l'ensemble des autres variables. On dépasse la simple corrélation ici. On le lit sur la diagonale de l'inverse de la matrice de corrélation (Régression, pages 53 et 54).

```
#critère VIF
vif = np.linalg.inv(mc)
print(vif)
```

Une variable pose problème dès lors que VIF > 4 (en étant très restrictif). Nous avons manifestement des problèmes dans nos données.

```
[[ 12.992577    -9.20146328  -3.7476397 ]
 [ -9.20146328   9.6964853    0.02124552]
 [ -3.7476397    0.02124552   4.26091058]]
```

Solutions possibles. Les techniques de régularisation permettent de dépasser les problèmes de colinéarité (ex. régression ridge, etc.). Statsmodels ne semble pas en proposer (j'ai bien cherché). De même, il n'existe pas, semble-t-il (là aussi j'ai bien cherché) de dispositif de sélection de variables. Ces absences ne constituent pas vraiment un problème à bien y regarder. Nous disposons d'un langage de programmation puissant avec Python. Il nous est très facile de programmer les techniques usuelles de régularisation et/ou de sélection. Avis aux amateurs (je les vois bien en sujet d'examen pour mes étudiants aussi, hum…).

7 Prédiction ponctuelle et par intervalle

La généralisation sur de nouveaux individus est une des principales finalités de l'analyse prédictive. L'idée est d'obtenir une valeur probable de la variable cible en appliquant le modèle sur des individus pour lesquels nous disposons des valeurs des variables exogènes.

Dans notre cas, il s'agit avant tout d'un exercice de style. Nous avions mis de côté un certain nombre d'observations au préalable. Nous disposons donc à la fois les valeurs des exogènes et mais aussi de l'endogène. Nous pourrions ainsi vérifier la fiabilité de notre modèle en confrontant les valeurs prédites et observées. C'est une procédure de validation externe en quelque sorte. Elle est très pratiquée dans la communauté machine learning (apprentissage supervisé), moins en économétrie[3].

[3] Durant mes études d'économétrie, je n'ai jamais entendu parler de l'idée d'appliquer en aveugle les modèles sur un échantillon test pour en mesurer les performances prédictives. Je me rends compte qu'on en parle un peu plus aujourd'hui dans les ouvrages, notamment en faisant référence à la validation croisée ou à des indicateurs de type PRESS.

7.1 Chargement du fichier

Le fichier « vehicules_2.txt » se présente comme suit.

modele	cylindree	puissance	poids	conso
Nissan Primera 2.0	1997	92	1240	9.2
Fiat Tempra 1.6 Liberty	1580	65	1080	9.3
Opel Omega 2.5i V6	2496	125	1670	11.3
Subaru Vivio 4WD	658	32	740	6.8
Seat Ibiza 2.0 GTI	1983	85	1075	9.5
Ferrari 456 GT	5474	325	1690	21.3

Pour ces n*=6 automobiles, nous devons calculer les prédictions en utilisant les coefficients estimés du modèle et les valeurs observées des exogènes. Ensuite, nous les comparerons avec les valeurs réelles de consommation pour en évaluer la qualité.

Nous chargeons les observations dans un nouveau data frame **cars2**.

```
#chargement du second fichier de données
cars2 = pandas.read_table("vehicules_2.txt",sep="\t",header=0,index_col=0)

#nombre d'obs. à prédire
n_pred = cars2.shape[0]

#liste des modèles
print(cars2)
```

La structure du fichier (index, colonnes) est identique à notre ensemble de données initial.

```
                        cylindree  puissance  poids  conso
modele
Nissan Primera 2.0           1997         92   1240    9.2
Fiat Tempra 1.6 Liberty      1580         65   1080    9.3
Opel Omega 2.5i V6           2496        125   1670   11.3
Subaru Vivio 4WD              658         32    740    6.8
Seat Ibiza 2.0 GTI           1983         85   1075    9.5
Ferrari 456 GT               5474        325   1690   21.3
```

7.2 Prédiction ponctuelle

La prédiction ponctuelle est obtenue en appliquant les coefficients estimés sur la description (valeurs des exogènes) de l'observation à traiter. Passer par une opération matricielle est conseillée lorsque le nombre d'observations à manipuler est élevé.

Pour un ensemble d'individus, la formule s'écrit (Régression, section 12.1, page 125) :

$$y_* = X_* \times \hat{a}$$

Où â est le vecteur des coefficients estimés, de dimension (p+1). Attention, pour que l'opération soit cohérente, il faut ajouter la constante (valeur 1) à la matrice X_* des n* individus à traiter.

Sous Python, nous devons tout d'abord former la matrice X_* en lui adjoignant la constante avec la commande **add_constant()** :

```
#exogènes du fichier
cars2_exog = cars2[['cylindree','puissance','poids']]

#ajouter une constante pour que le produit matriciel fonctionne
cars2_exog = sm.add_constant(cars2_exog)
print(cars2_exog)
```

L'endogène ne fait pas partie de la matrice X_* bien évidemment.

```
                        const  cylindree  puissance  poids
modele
Nissan Primera 2.0          1       1997         92   1240
Fiat Tempra 1.6 Liberty     1       1580         65   1080
Opel Omega 2.5i V6          1       2496        125   1670
Subaru Vivio 4WD            1        658         32    740
Seat Ibiza 2.0 GTI          1       1983         85   1075
Ferrari 456 GT              1       5474        325   1690
```

Il ne reste plus qu'à appliquer les coefficients du modèle avec **predict**()…

```
#réaliser la prédiction ponctuelle - reg est l'objet régression
pred_conso = reg.predict(res.params,cars2_exog)
print(pred_conso)
```

… pour obtenir les prédictions ponctuelles.

```
[  9.58736855   8.1355859   12.58619059   5.80536066   8.52394228  17.33247145]
```

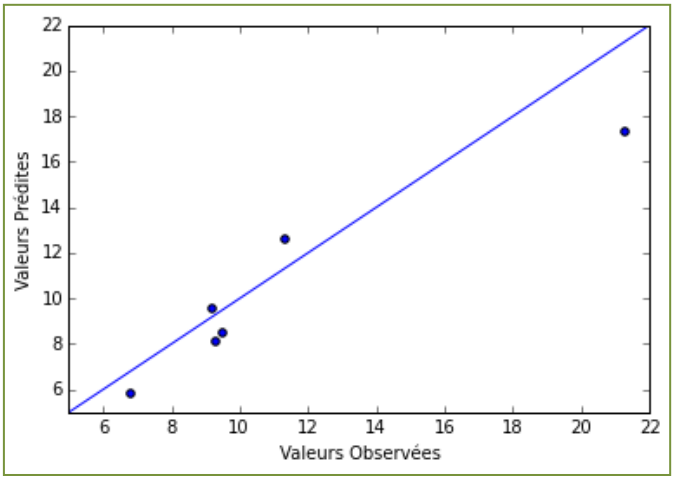
7.3 Confrontation : valeurs prédites vs. observées

Un graphique permet de vérifier la qualité de prédiction en un coup d'œil. En abscisse, nous mettrons les valeurs réelles, en ordonnées les prédictions. Dans l'idéal, si le modèle est parfait, les points seraient situés sur la diagonale. Nous utilisons le package **matplotlib** pour confectionner le nuage de points.

```
#confrontation obs. vs. pred.
import matplotlib.pyplot as plt
plt.scatter(cars2['conso'],pred_conso)
plt.plot(np.arange(5,23),np.arange(5,23))
plt.xlabel('Valeurs Observées')
plt.ylabel('Valeurs Prédites')
```

```
plt.xlim(5,22)
plt.ylim(5,22)
plt.show()
```

La prédiction est plus ou moins crédible pour 5 points. Pour le 6ème, la Ferrari, le modèle sous-estime fortement la consommation (17.33 vs. 21.3).



7.4 Prédiction par intervalle

Une prédiction ponctuelle donne un ordre d'idées. Une prédiction par intervalle est toujours préférable parce que l'on peut associer un risque (de se tromper) à la prise de décision.

Variance de l'erreur de prédiction. Le calcul est assez complexe (Régression, section 12.2, page 126). Elle nécessite le calcul de la variance de l'erreur de prédiction. Pour l'individu i* (équation 12.2), la formule s'écrit :

$$\hat{\sigma}^2_{\hat{\varepsilon}_{i*}} = \hat{\sigma}^2_{\varepsilon}\left[1 + X_{i*}\left(X'X\right)^{-1} X'_{i*}\right]$$

Où certaines informations sont extraites du modèle estimé et directement accessibles avec les objets **reg** et **res** (sections 4.1 et 4.2) : $\hat{\sigma}^2_{\varepsilon}$ est la variance de l'erreur, $\left(X'X\right)^{-1}$ est issue des données ayant servi à élaboré le modèle. D'autres informations sont relatives à l'individu i* à traiter : X_{i*} est la description de l'individu incluant la constante. Par exemple, pour le premier individu (Nissan Primera 2.0), nous aurons le vecteur (1 ; 1997 ; 92 ; 1240).

Voyons comment articuler tout cela dans un programme Python.

```
#récupération de la matrice (X'X)-1
inv_xtx = reg.normalized_cov_params
print(inv_xtx)
```

```
#transformation en type matrice des exogènes à prédire
X_pred = cars2_exog.as_matrix()

#### variance de l'erreur de prédiction ###
#initialisation
var_err = np.zeros((n_pred,))
#pour chaque individu à traiter
for i in range(n_pred):
    #description de l'indiv.
    tmp = X_pred[i,:]
    #produit matriciel
    pm = np.dot(np.dot(tmp,inv_xtx),np.transpose(tmp))
    #variance de l'erreur
    var_err[i] = res.scale * (1 + pm)
#
print(var_err)
```

Nous avons :

```
[ 0.51089413  0.51560511  0.56515759  0.56494062  0.53000431  1.11282003]
```

Intervalle de confiance. Il ne nous reste plus qu'à calculer les bornes basses et hautes de l'intervalle de prédiction à 95% (équation 12.4) en exploitant la prédiction ponctuelle et le quantile de la loi de Student.

```
#quantile de la loi de Student pour un intervalle à 95%
qt = scipy.stats.t.ppf(0.975,df=n-p-1)

#borne basse
yb = pred_conso - qt * np.sqrt(var_err)
print(yb)

#borne haute
yh = pred_conso + qt * np.sqrt(var_err)
print(yh)
```

Nous avons respectivement :

```
[  8.1009259    6.6423057   11.02280004   4.24227024   7.00995439  15.13868086]
[ 11.07381119   9.62886611  14.14958114   7.36845108  10.03793016  19.52626203]
```

Confrontation avec les valeurs réelles. La confrontation prend une dimension concrète ici (on ne juge plus sur une impression) puisqu'il s'agit de vérifier que les fourchettes contiennent la vraie valeur de l'endogène. Nous organisons la présentation de manière à faire encadrer les valeurs observées par les bornes basses et hautes des intervalles.

```
#matrice collectant les différentes colonnes (yb,yobs,yh)
a = np.resize(yb,new_shape=(n_pred,1))
y_obs = cars2['conso']
a = np.append(a,np.resize(y_obs,new_shape=(n_pred,1)),axis=1)
a = np.append(a,np.resize(yh,new_shape=(n_pred,1)),axis=1)

#mettre sous forme de data frame pour commodité d'affichage
df = pandas.DataFrame(a)
df.index = cars2.index
df.columns = ['B.Basse','Y.Obs','B.Haute']
print(df)
```

C'est confirmé, la Ferrari dépasse les bornes (si je puis dire…).

```
                          B.Basse  Y.Obs    B.Haute
modele
Nissan Primera 2.0        8.100926    9.2  11.073811
Fiat Tempra 1.6 Liberty   6.642306    9.3   9.628866
Opel Omega 2.5i V6       11.022800   11.3  14.149581
Subaru Vivio 4WD          4.242270    6.8   7.368451
Seat Ibiza 2.0 GTI        7.009954    9.5  10.037930
Ferrari 456 GT           15.138681   21.3  19.526262
```

8 Conclusion

Le package StatsModels offre nativement des fonctionnalités intéressantes pour la modélisation statistique. Couplé avec le langage Python et les autres packages (numpy, scipy, pandas, etc.), le champ des possibilités devient immense. On retrouve très facilement ici les reflexes et les compétences que nous avons pu développer sous R. Le plus dur finalement reste l'accès à la documentation. J'ai du batailler pour trouver la bonne commande à chaque étape. Les tutoriels en français sont quasiment inexistants à ce jour (fin septembre 2015). On va essayer d'arranger cela.