1 Objectif

Un cas d'étude pour montrer les pistes d'optimisation d'un programme sous Python. Outils de débogage, d'analyse et de profilage de code sous l'éditeur SPYDER. Programmation de la procédure ''leave-one-out'' pour l'évaluation des modèles en analyse prédictive.

J'avais écrit récemment un document à propos de l'optimisation des programmes sous R (''Programmer efficacement sous R'', février 2019). Dans ce tutoriel, nous étudierons cette fois-ci comment déboguer, analyse et optimiser du code en Python, via l'EDI (Environnement de Développement Intégré) SPYDER livré avec la distribution ANACONDA.

D'autres environnements de développements existent pour Python (''Here are the most popular Python IDEs / Editors'', KDnuggets, Décembre 2018) mais, pour ma part, SPYDER me convient très bien au jour le jour. Je le conseille souvent à mes étudiants, en partie à cause de sa similitude avec RStudio. L'interface leur étant familière, le passage d'un langage à l'autre est moins abrupt.

Tout comme pour R, nous prétexterons de l'implémentation du leave-one-out (LOOCV – Leave-One-Out Cross-Validation) en modélisation prédictive (analyse discriminante linéaire) pour explorer les fonctionnalités proposées par SPYDER.

2 Procédure ''leave-one-out''

Le ''leave-one-out'' (**LOOCV**, leave-one-out cross-validation) est une procédure de rééchantillonnage qui permet d'estimer les performances prédictives d'un modèle élaboré sur un dataset de taille n. Cette technique est un cas particulier de la validation croisée, qui s'impose lorsque nous disposons d'une base de données de taille réduite et qu'il n'est pas réaliste de la subdiviser en échantillons d'apprentissage et de test.

L'algorithme est le suivant :

v = vecteur des prédictions

Pour i allant de 1 à n (taille de la base de données)

    Construire le modèle sur toutes les observations sauf le n°i (n-1 observations)

    Prédire sur l'individu n°i, collecter la valeur dans v

Croiser l'attribut cible et le vecteur des prédictions v

En déduire la proportion de mauvaises prédictions, il s'agit du taux d'erreur en LOOCV

On va construire n fois un modèle prédictif, mieux vaut avoir une bonne machine et/ou savoir programmer astucieusement. C'est ce que nous allons essayer de faire dans ce tutoriel, en nous aidant des outils de profiling.

Nous utiliserons l'analyse discriminante (LinearDiscriminantAnalysis du package ''scikit-learn'') en guise d'algorithme de machine learning. L'information n'est pas anodine. En effet, elle est stable (faible variance), peu propice au surapprentissage, l'utilisation de la LOOCV pour mesurer l'erreur en généralisation se justifie pleinement dans son contexte. C'est un peu moins vrai lorsque nous avons affaire à des algorithmes qui ont tendance à trop coller aux données.

3 Données

Nous utilisons une variante de la base ''waveform'' avec 1000 observations. La variable cible ''onde'' est à 3 modalités, nous disposons de 21 descripteurs (V1...V21). De fait, nous aurons à réaliser n = 1000 cycles d'apprentissage sur (n – 1) = 999 observations durant la LOOCV. Il faut espérer que la classe LinearDiscriminantAnalysis de 'scikit-learn'' soit programmée efficacement, sinon nous risquons d'attendre un moment devant notre PC.

Voici le code pour importer et vérifier l'intégrité des données.

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

#chargement des données
import pandas
wave = pandas.read_csv("wave1000.txt", sep="\t", header=0)
print(wave.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 22 columns):
onde    1000 non-null object
V01     1000 non-null float64
V02     1000 non-null float64
V03     1000 non-null float64
V04     1000 non-null float64
V05     1000 non-null float64
V06     1000 non-null float64
V07     1000 non-null float64
V08     1000 non-null float64
V09     1000 non-null float64
V10     1000 non-null float64
V11     1000 non-null float64
```

```
v12     1000 non-null float64
v13     1000 non-null float64
v14     1000 non-null float64
v15     1000 non-null float64
v16     1000 non-null float64
v17     1000 non-null float64
v18     1000 non-null float64
v19     1000 non-null float64
v20     1000 non-null float64
v21     1000 non-null float64
dtypes: float64(21), object(1)
memory usage: 172.0+ KB
```

4 Première version du leave-one-out en Python

4.1 Implémentation et mesure du temps d'exécution

Plusieurs éléments sont nécessaires pour implémenter la LOOCV. Détaillons-les.

Compteur de durée. Tout d'abord, pour mettre en place un compteur de durée d'exécution, nous récupérons la valeur courante de l'horloge de la machine (en durée de secondes depuis la date d'initialisation du compteur).

```
#pour mesurer le temps
import time
debut = time.time()
```

Vecteur de prédiction. Pour les collecter les prédictions sur les $i^{ème}$ individus successifs de la LOOCV, nous initialisons un vecteur vide.

```
#vecteur de prédiction
import numpy
prediction = numpy.array([], dtype='object')
```

Outil pour la LOOCV. Pour définir les sous-échantillons de la LOOCV à chaque itération, nous instancions un objet de la classe LeaveOneOut de ''scikit-learn''. Elle se chargera de produire les indices des individus concernés par l'apprentissage et la prédiction au moment voulu.

```
#classe pour leave one out
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
```

Classe pour la modélisation. Enfin, nous importons la classe LinearDiscriminantAnalysis de ''scikit-learn'' pour la modélisation à l'aide l'analyse discriminante. Nous nous appuierons sur les paramètres par défaut de l'implémentation proposée (version **0.20.3** de sklearn).

```
#algo. pour modélisation
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

Boucle de calcul de la LOOCV. Voici la boucle de calcul de la LOOCV.

```
#itérer sur les observations en apprentissage et en test
for train_index, test_index in loo.split(wave.iloc[:,1:]):
    #instancier l'algo
    lda = LinearDiscriminantAnalysis()
    #modéliser
    lda.fit(X=wave.iloc[train_index,1:], y=wave.iloc[train_index,0])
    #prédire
    p = lda.predict(X=wave.iloc[test_index,1:])
    #ajouter dans les résultats
    prediction = numpy.append(prediction, p)
```

Nous distinguons :

- La boucle ***for*** qui itère sur les index des observations en apprentissage (train_index, une liste avec 999 valeurs) et en test (test_index, une liste avec une seule valeur) fournis par *loo.split()*.
- Puis, à chaque itération, nous créons une instance de l'analyse discriminante en faisant appel au constructeur de *LinearDiscriminantAnalysis()*.
- Les coefficients de l'analyse discriminante sont calculés à l'aide de la fonction *fit()* appliquée au données d'apprentissage c.-à-d. au data.frame indexé par les individus appartenant à l'échantillon d'apprentissage.
- Nous effectuons la prédiction sur l'unique observation en test avec *predict()*.
- Enfin, nous récupérons cette prédiction en l'ajoutant *append()* dans le vecteur que nous avions préparé à cet effet plus haut.

L'opération la plus coûteuse est la phase de modélisation à partir de données avec *fit()* qui sera répétée n = 1000 fois et, malheureusement, il n'est pas possible de jouer dessus, sauf à modifier les paramètres de l'instance lors de son initialisation (lors de l'appel du constructeur). Mais ceci est un autre sujet qui dépasse le cadre de l'optimisation du programme Python puisque nous touchons aux propriétés mêmes de l'algorithme d'apprentissage.

Calcul du taux de reconnaissance. Nous calculons le taux de succès en faisant appel à la fonction *accuracy_score()* de ''scikit-learn''.

```
#afficher le taux de bonnes prédictions
from sklearn.metrics import accuracy_score
print('Acc. score = ' + str(accuracy_score(wave.onde, prediction)))
```

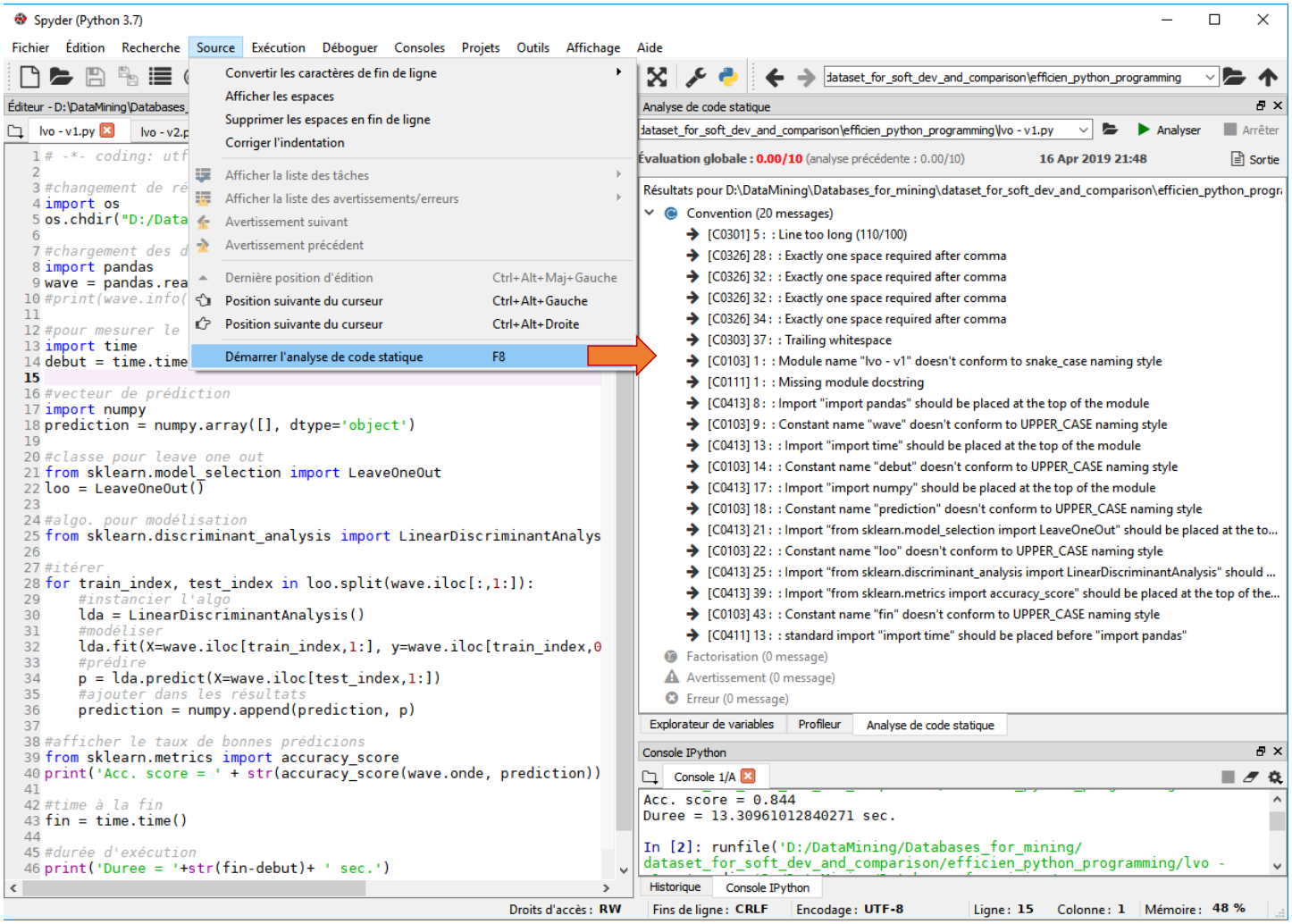
Durée de calcul. Enfin, nous calculons le temps de traitements en faisant de nouveau appel à time(). Nous déduisons la durée via l'écart avec la valeur obtenue lors de l'appel précédent.

```
#time à la fin
fin = time.time()

#durée d'exécution
print('Duree = '+str(fin-debut)+ ' sec.')
```

Pour cette première version de notre implémentation, la durée d'exécution de la validation croisée est d'approximativement **7.25 secondes**. Je dis bien approximativement parce que nous obtenons une valeur légèrement différente à chaque lancement du programme.

4.2 Analyse statique du code

''Est-ce que mon programme est bien écrit ?'' est une interrogation qui taraude toujours tout bon programmeur. SPYDER propose un analyseur de code. Nous y avons accès via le menu SOURCE / DEMARRER L'ANALYSE DE CODE STATIQUE.

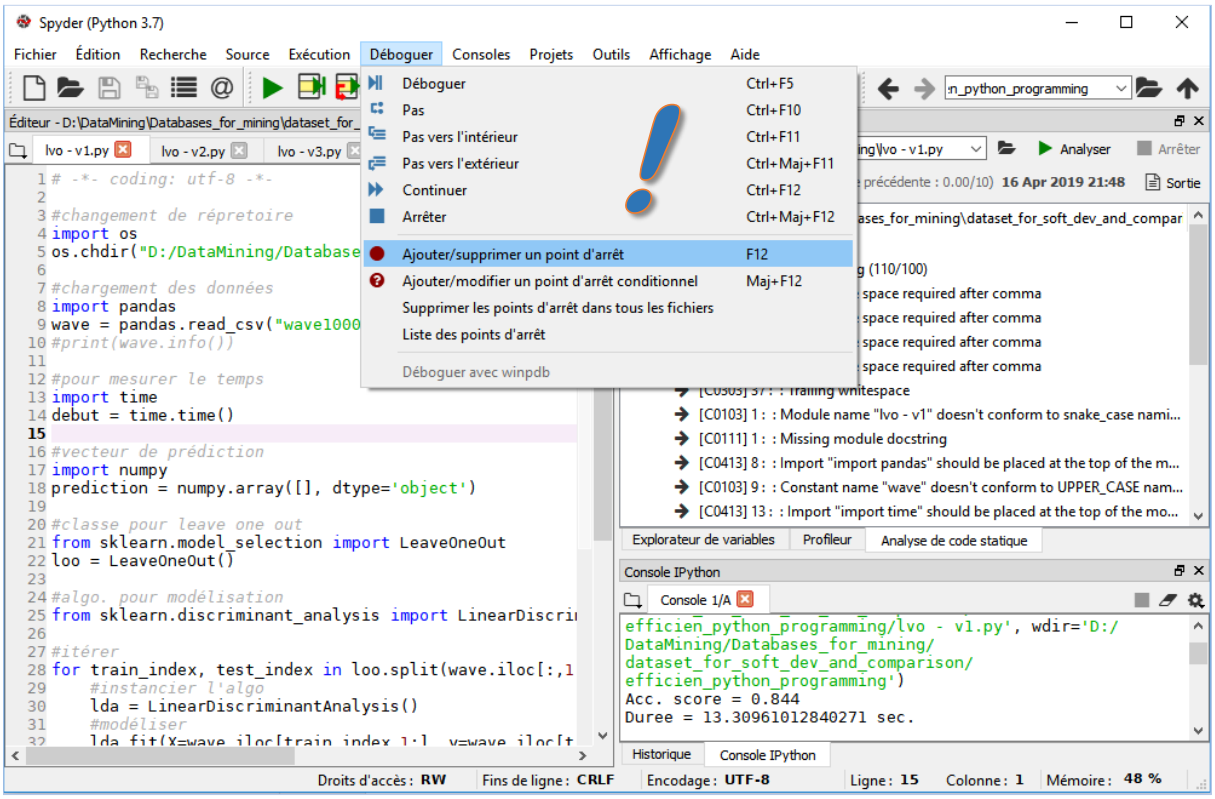


J'ai la note de **0.00/10** (en rouge dans la fenêtre de droite). Bien ! C'est encourageant. Il semble surtout que je ne sois pas très attentif aux conventions d'écriture : il manque l'espace derrière certaines virgules (lignes 28, 32, etc.), les importations de modules (**import**) ne sont pas regroupées dans la partie haute du programme (ex. pandas, time, numpy, etc.), je ne respecte pas les règles de nommage des variables, etc. Rien de bien méchant à vrai dire. Il n'y a ni warning, ni erreurs. On s'en contentera. En tous les cas, ce type d'outil permet d'acquérir les bons réflexes. Les règles de nommage des variables par exemple ne sont pas très importants tant que l'on travaille tout seul, ils le deviennent dès lors que l'on travaille en équipe et qu'il faut partager du code avec d'autres programmeurs.

4.3 Débogage du programme

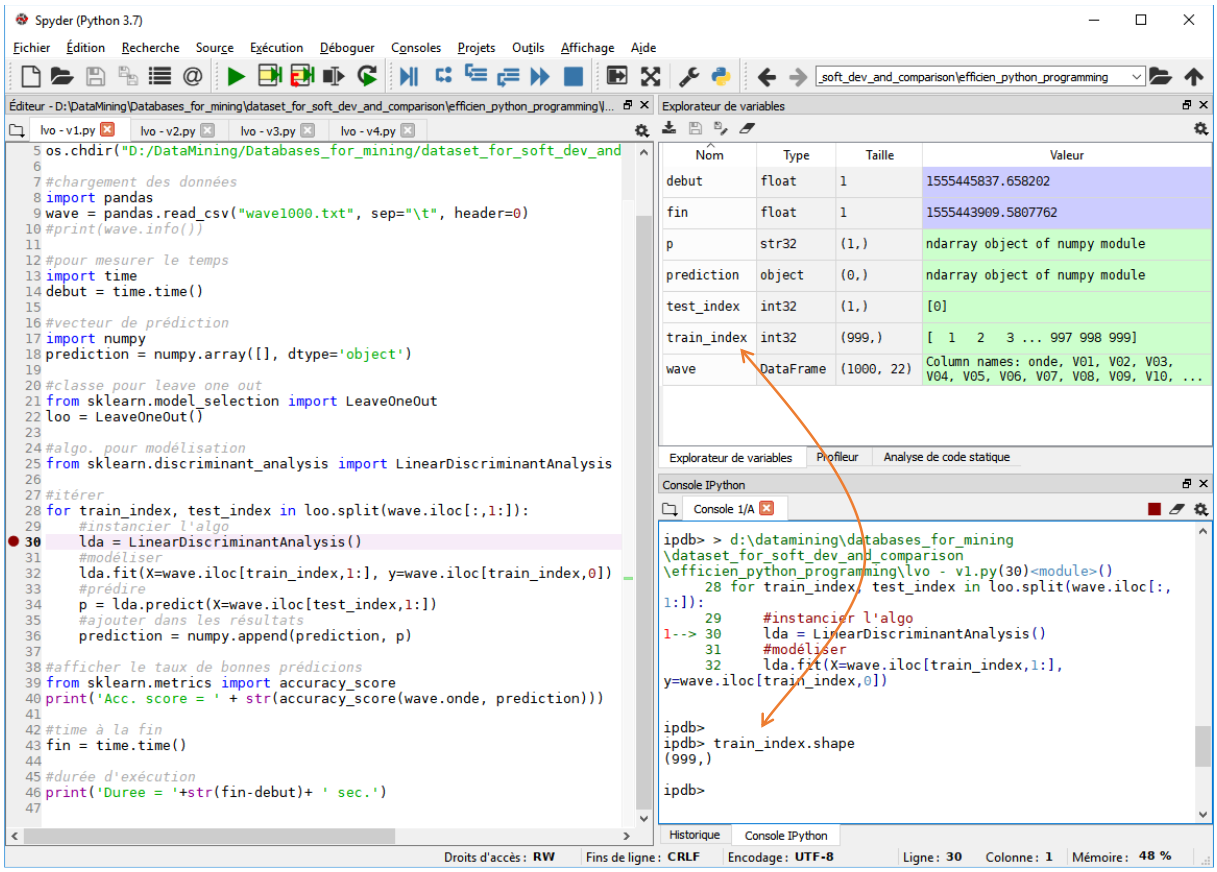
Suivre pas-à-pas le déroulement du programme est essentiel pour en comprendre le fonctionnement, en surveillant les valeurs prises par les différentes variables, en suivant le chemin suivi par l'exécution selon le contexte d'utilisation. Truffer son code de print() est une solution simple et immédiate, mais ingérable à mesure que notre code augmente de volume. Il faut alors passer par le débogueur intégré de l'EDI.

L'outil DEBOGUEUR de SPYDER est très intéressant dans cette optique.

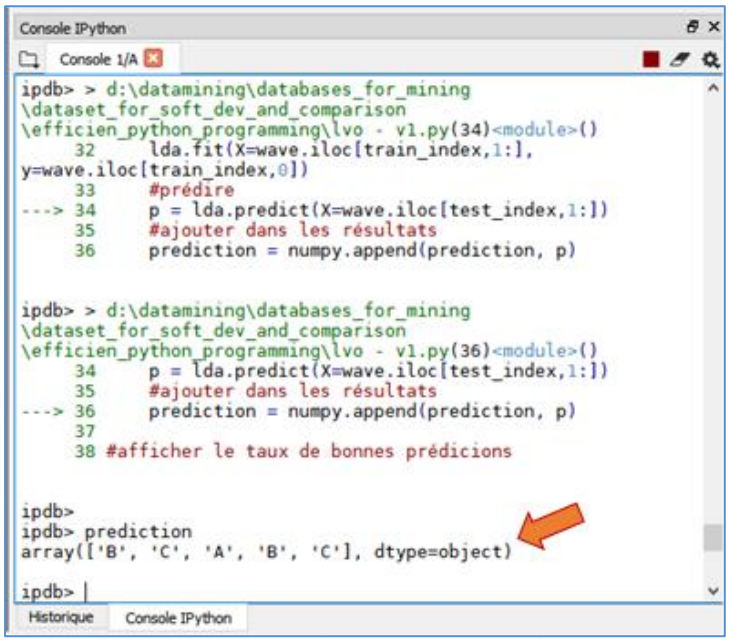


Par exemple, nous pouvons ajouter un point d'arrêt à la ligne 30 de notre programme (rond rouge dans l'EDI à la ligne 30 ci-dessous), juste avant l'instanciation de l'algorithme de machine learning.

A l'exécution (menu DEBOGUER / DEBOGUER), nous observons dans l'EDI (*le programme est arrêté à la ligne 30 dans la copie d'écran ci-dessous, elle est légèrement surlignée*) :



Nous visualisons dans l'**explorateur de variables** les valeurs courantes des objets. Et dans la **console**, nous pouvons approfondir nos investigations. Par exemple, je demande ici les dimensions du vecteur train_index. Il comporte 999 valeurs.



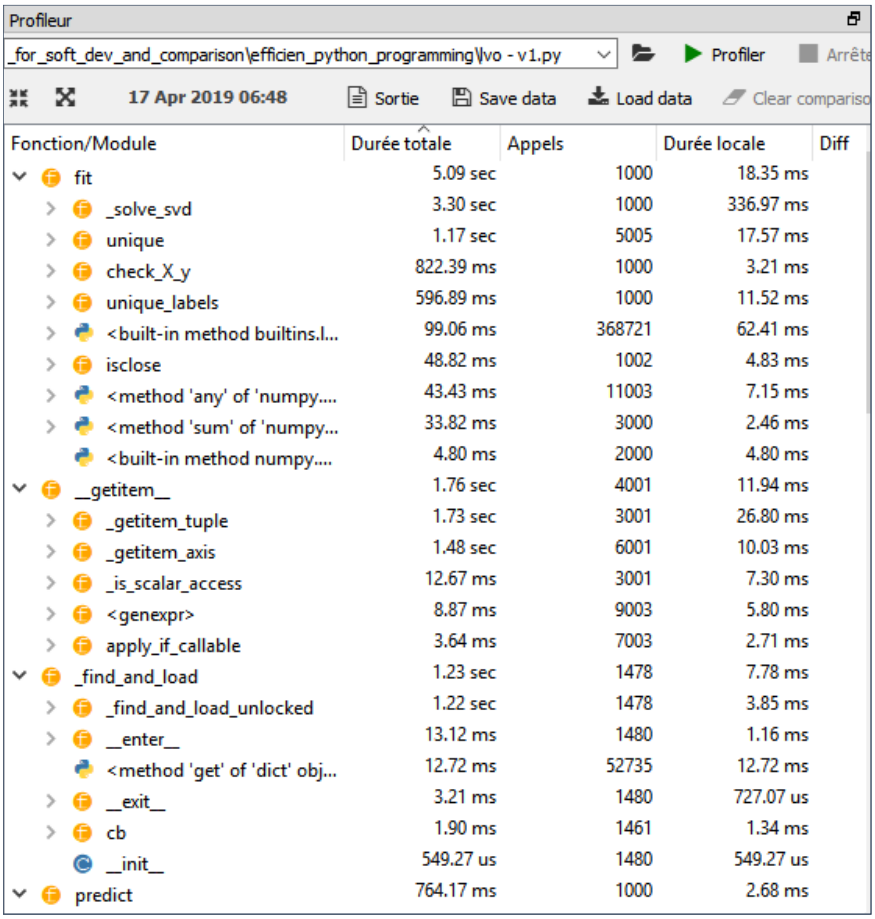
Nous pouvons par la suite poursuivre l'exécution ligne à ligne du programme (DEBOGUER / PAS) et surveiller l'état des variables. Après 5 itérations, voici ci-contre ce que nous observons dans le vecteur des prédictions via la console.

Nous pouvons terminer l'exécution du programme

en retirant le point d'arrêt et en cliquant sur le menu DEBOGUER / CONTINUER.

4.4 Profilage du code

Venons-en maintenant au thème principal de ce tutoriel, l'optimisation de notre programme via l'outil de profilage fourni par SPYDER. Nous cliquons sur le menu EXECUTION / PROFILER. Voici ce que nous avons dans la fenêtre PROFILEUR dans un premier temps.

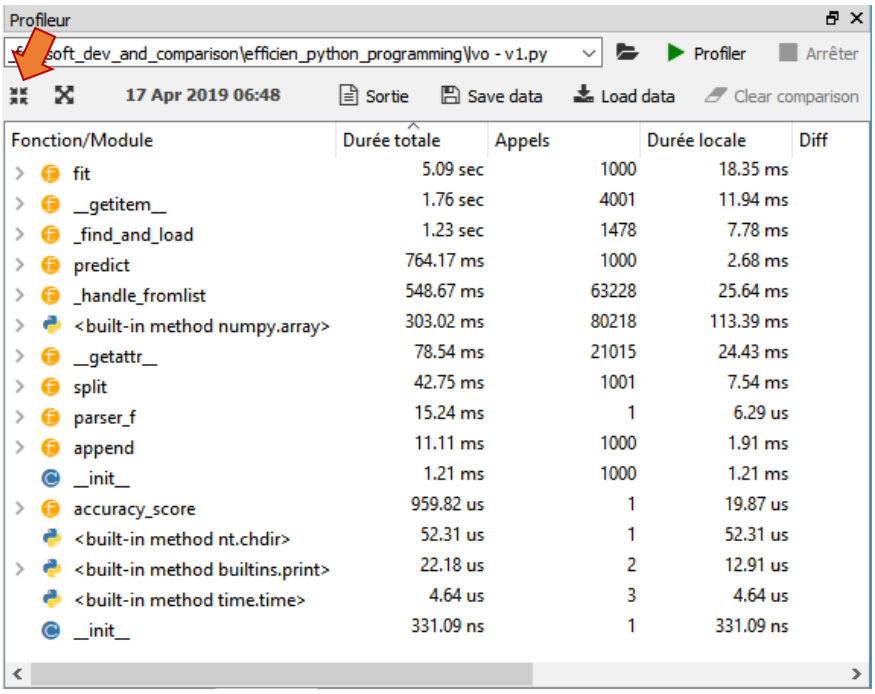


Fonction/Module	Durée totale	Appels	Durée locale	Diff
fit	5.09 sec	1000	18.35 ms	
_solve_svd	3.30 sec	1000	336.97 ms	
unique	1.17 sec	5005	17.57 ms	
check_X_y	822.39 ms	1000	3.21 ms	
unique_labels	596.89 ms	1000	11.52 ms	
<built-in method builtins.I...	99.06 ms	368721	62.41 ms	
isclose	48.82 ms	1002	4.83 ms	
<method 'any' of 'numpy....	43.43 ms	11003	7.15 ms	
<method 'sum' of 'numpy...	33.82 ms	3000	2.46 ms	
<built-in method numpy....	4.80 ms	2000	4.80 ms	
__getitem__	1.76 sec	4001	11.94 ms	
_getitem_tuple	1.73 sec	3001	26.80 ms	
_getitem_axis	1.48 sec	6001	10.03 ms	
_is_scalar_access	12.67 ms	3001	7.30 ms	
<genexpr>	8.87 ms	9003	5.80 ms	
apply_if_callable	3.64 ms	7003	2.71 ms	
_find_and_load	1.23 sec	1478	7.78 ms	
_find_and_load_unlocked	1.22 sec	1478	3.85 ms	
__enter__	13.12 ms	1480	1.16 ms	
<method 'get' of 'dict' obj...	12.72 ms	52735	12.72 ms	
__exit__	3.21 ms	1480	727.07 us	
cb	1.90 ms	1461	1.34 ms	
__init__	549.27 us	1480	549.27 us	
predict	764.17 ms	1000	2.68 ms	

Plutôt que par ligne de code, les temps de traitements sont regroupés par fonctions appelées, lesquelles étant décomposées en appels de sous-fonctions. Elles sont triées par durée de traitements. Nous disposons également du nombre d'appels. Cette information est très importante. Gagner ne serait-ce qu'une infime fraction de secondes sur une instruction qui est appelée un très grand nombre de fois fera gagner un temps considérable à l'heure du bilan.

Il est possible de sauvegarder ces informations en cliquant sur le bouton SAVE DATA, nous créons le fichier ''**lvo – v1.Result**'' (c'est un fichier binaire qu'on ne peut pas lire dans un éditeur de texte).

La description est trop détaillée dans cette présentation. Nous revenons au niveau le plus élevé en cliquant sur l'icone la plus à gauche (les 4 flèches qui se rejoignent).



Fonction/Module	Durée totale	Appels	Durée locale	Diff
fit	5.09 sec	1000	18.35 ms	
__getitem__	1.76 sec	4001	11.94 ms	
_find_and_load	1.23 sec	1478	7.78 ms	
predict	764.17 ms	1000	2.68 ms	
_handle_fromlist	548.67 ms	63228	25.64 ms	
<built-in method numpy.array>	303.02 ms	80218	113.39 ms	
__getattr__	78.54 ms	21015	24.43 ms	
split	42.75 ms	1001	7.54 ms	
parser_f	15.24 ms	1	6.29 us	
append	11.11 ms	1000	1.91 ms	
__init__	1.21 ms	1000	1.21 ms	
accuracy_score	959.82 us	1	19.87 us	
<built-in method nt.chdir>	52.31 us	1	52.31 us	
<built-in method builtins.print>	22.18 us	2	12.91 us	
<built-in method time.time>	4.64 us	3	4.64 us	
__init__	331.09 ns	1	331.09 ns	

Revoyons notre code source pour identifier chaque item de cette liste.

```
1  # -*- coding: utf-8 -*-
2
3  #changement de répretoire
4  import os
5  os.chdir("D:/DataMining/Databases_for_mining/dataset_for_soft_dev_and_c
6
7  #chargement des données
8  import pandas
9  wave = pandas.read_csv("wave1000.txt", sep="\t", header=0)
10 #print(wave.info())
11
12 #pour mesurer le temps
13 import time
14 debut = time.time()
15
16 #vecteur de prédiction
17 import numpy
18 prediction = numpy.array([], dtype='object')
19
20 #classe pour leave one out
21 from sklearn.model_selection import LeaveOneOut
22 loo = LeaveOneOut()
23
24 #algo. pour modélisation
25 from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
26
27 #itérer
28 for train_index, test_index in loo.split(wave.iloc[:,1:]):
29     #instancier l'algo
30     lda = LinearDiscriminantAnalysis()
31     #modéliser
32     lda.fit(X=wave.iloc[train_index,1:], y=wave.iloc[train_index,0])
33     #prédire
34     p = lda.predict(X=wave.iloc[test_index,1:])
35     #ajouter dans les résultats
36     prediction = numpy.append(prediction, p)
37
38 #afficher le taux de bonnes prédicions
39 from sklearn.metrics import accuracy_score
40 print('Acc. score = ' + str(accuracy_score(wave.onde, prediction)))
41
42 #time à la fin
43 fin = time.time()
44
45 #durée d'exécution
46 print('Duree = '+str(fin-debut)+ ' sec.')
```

Si l'on se focalise sur les fonctions les plus gourmandes :

- Les instructions fit() (ligne 32, 5.09 sec.), predict() (ligne 34, 764.17 ms.) et _init_ (ligne 30, 1.21 ms.), répétées 1000 fois, sont faciles à identifier. Gagner du temps sur les deux premières instructions n'est pas trop possibles, sur la troisième, nous verrons plus loin qu'il est possible de la sortir de la boucle mais le gain est négligeable.
- Le _getitem_ appelé 4001 fois paraît mystérieux (1.76 sec.). En scrutant le code, on se rend compte qu'il s'agit des accès au data.frame ''wave'' (objet ''pandas'') que l'on observe dans les lignes n°28, 32 (2 fois) et 34.
- _find_and_load (1.23 sec., 1478 appels) correspond apparemment aux mécanismes d'accès internes aux objets. J'ai un peu (beaucoup) regardé. Nous n'avons pas vraiment prise là-dessus. Les opportunités d'optimisation ne sont pas évidentes.

Dans les sections suivantes, nous essayons de réduire le temps de calcul en jouant sur la réorganisation des commandes et les choix de structure de données.

5 Seconde version – Passage aux matrices Numpy

Tout comme sous R, je suspecte que les data.frame (''pandas'' ici, basé sur des listes) sont moins performantes que les matrices (''numpy'' sous Python). Nous créons deux structures intermédiaires pour la matrice des descripteurs (Xdata) et le vecteur de la variable cible (Ydata).

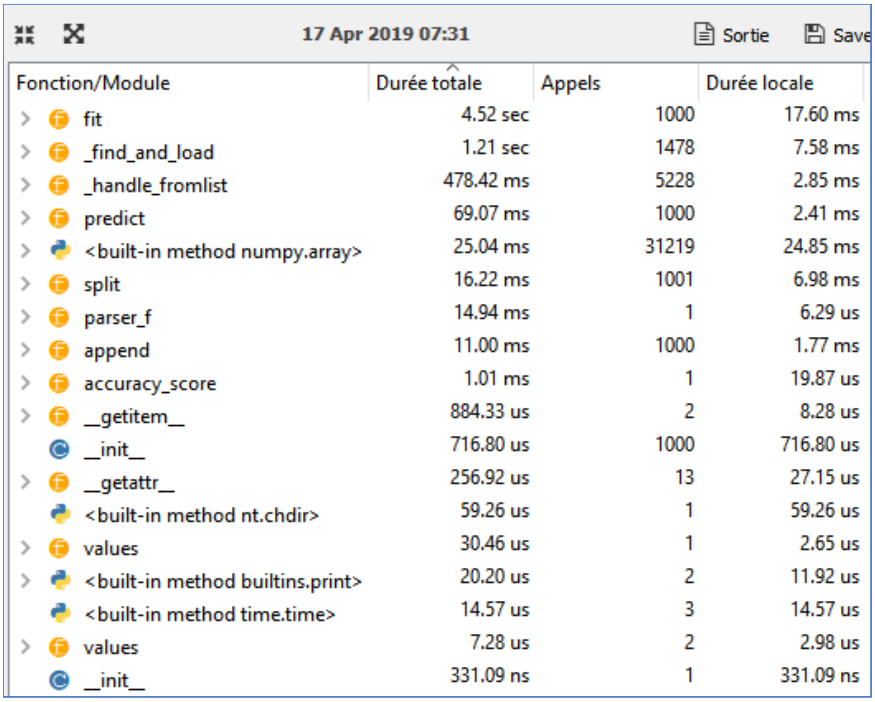
Voici la nouvelle version du code source :

```
#pour mesurer le temps
import time
debut = time.time()

#vecteur de prédiction
import numpy
prediction = numpy.array([], dtype='object')

#classe pour leave one out
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()

#algo. pour modélisation
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

#matrice et vecteur de données
Xdata = wave.iloc[:,1:].values
Ydata = wave.iloc[:,0].values
```

```
#itérer
for train_index, test_index in loo.split(Xdata):
    #instancier l'algo
    lda = LinearDiscriminantAnalysis()
    #modéliser
    lda.fit(X=Xdata[train_index,:], y=Ydata[train_index])
    #prédire
    p = lda.predict(X=Xdata[test_index,:])
    #ajouter dans les résultats
    prediction = numpy.append(prediction, p)

#afficher le taux de bonnes prédictions
from sklearn.metrics import accuracy_score
print('Acc. score = ' + str(accuracy_score(wave.onde, prediction)))

#fin
fin = time.time()

#durée
print('Duree = '+str(fin-debut)+ ' sec.')
```

Le temps de calcul passe à **4.6 sec.** en moyenne. Le gain est énorme (nous étions à 7.25 sec. dans la version précédente) mais il ne m'étonne pas. Nous avions observé le même phénomène sous R.

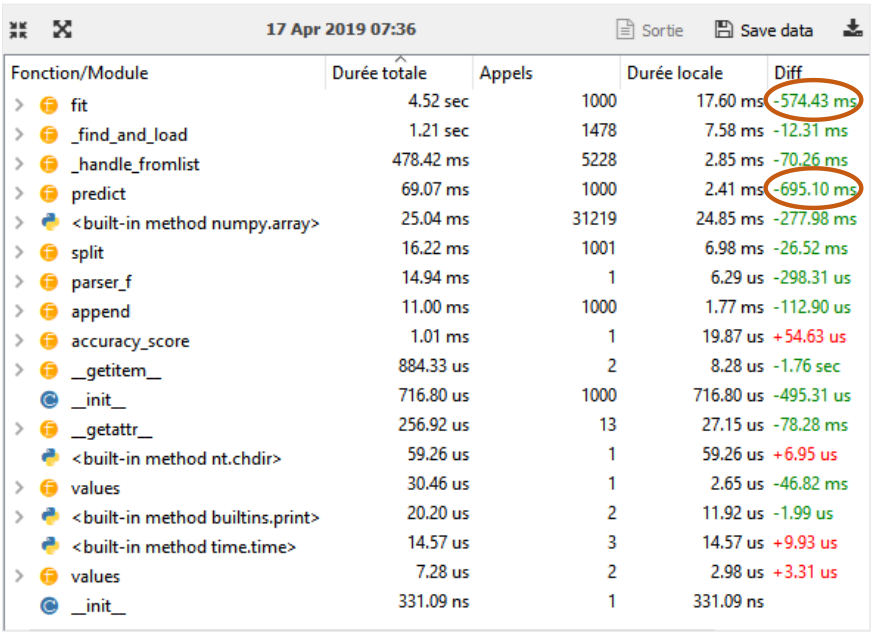
Un nouvel appel du profileur nous donne le tableau suivant :



Fonction/Module	Durée totale	Appels	Durée locale
fit	4.52 sec	1000	17.60 ms
_find_and_load	1.21 sec	1478	7.58 ms
_handle_fromlist	478.42 ms	5228	2.85 ms
predict	69.07 ms	1000	2.41 ms
<built-in method numpy.array>	25.04 ms	31219	24.85 ms
split	16.22 ms	1001	6.98 ms
parser_f	14.94 ms	1	6.29 us
append	11.00 ms	1000	1.77 ms
accuracy_score	1.01 ms	1	19.87 us
__getitem__	884.33 us	2	8.28 us
__init__	716.80 us	1000	716.80 us
__getattr__	256.92 us	13	27.15 us
<built-in method nt.chdir>	59.26 us	1	59.26 us
values	30.46 us	1	2.65 us
<built-in method builtins.print>	20.20 us	2	11.92 us
<built-in method time.time>	14.57 us	3	14.57 us
values	7.28 us	2	2.98 us
__init__	331.09 ns	1	331.09 ns

L'accès à la structure ''pandas'' matérialisée par _getitem_ répétés a disparu de la liste des instructions (il en reste 2, lorsqu'on crée les variables Xdata et Ydata).

Pour mieux comparer les différences, nous pouvons charger les informations de la précédente version ''lvo- v1.Result'' avec le bouton LOAD DATA.



Fonction/Module	Durée totale	Appels	Durée locale	Diff
fit	4.52 sec	1000	17.60 ms	-574.43 ms
_find_and_load	1.21 sec	1478	7.58 ms	-12.31 ms
_handle_fromlist	478.42 ms	5228	2.85 ms	-70.26 ms
predict	69.07 ms	1000	2.41 ms	-695.10 ms
<built-in method numpy.array>	25.04 ms	31219	24.85 ms	-277.98 ms
split	16.22 ms	1001	6.98 ms	-26.52 ms
parser_f	14.94 ms	1	6.29 us	-298.31 us
append	11.00 ms	1000	1.77 ms	-112.90 us
accuracy_score	1.01 ms	1	19.87 us	+54.63 us
__getitem__	884.33 us	2	8.28 us	-1.76 sec
__init__	716.80 us	1000	716.80 us	-495.31 us
__getattr__	256.92 us	13	27.15 us	-78.28 ms
<built-in method nt.chdir>	59.26 us	1	59.26 us	+6.95 us
values	30.46 us	1	2.65 us	-46.82 ms
<built-in method builtins.print>	20.20 us	2	11.92 us	-1.99 us
<built-in method time.time>	14.57 us	3	14.57 us	+9.93 us
values	7.28 us	2	2.98 us	+3.31 us
__init__	331.09 ns	1	331.09 ns	

Nous constatons, entre autres, que fit() et predict() bénéficient également de ce changement de structure de données.

6 Troisième version – Vecteur de booléens

On peut s'amuser longtemps comme cela. Dans cette dernière version, j'ai essayé de réduire encore ce qui était répétitif dans la boucle, à savoir les instructions split(), append() et l'instanciation de l'objet de calcul.

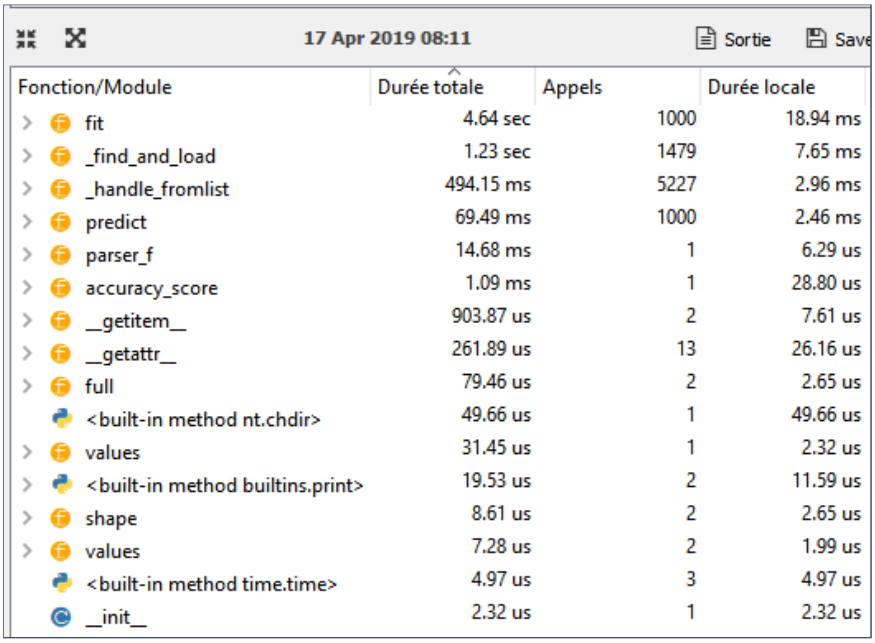
- Pour ce dernier la solution est simple, il suffit de la sortit de la boucle simplement.
- Pour split(), plutôt que les index d'observations fournis par la classe LeaveOneOut de ''scikit-learn'', nous utilisons un vecteur de booléens (index) pour désigner les individus en apprentissage et en test. A chaque itération, la valeur de l'individu exclu de l'apprentissage est passé à False.
- Pour éviter append() et son cortège de petites allocations mémoire successives, nous initialisons le vecteur de prédictions à la bonne taille dès le départ, puis nous la remplissons par accès indicé.

Voici la nouvelle version du code :

```
#pour mesurer le temps
import time
debut = time.time()

#vecteur de prédiction -- initialisation à la bonne taille
import numpy
prediction = numpy.full(wave.shape[0]," ", dtype='object')

#vecteur indiquant l'individu à traiter
index = numpy.full(wave.shape[0],True)

#algo. pour modélisation
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

#matrices
Xdata = wave.iloc[:,1:].values
Ydata = wave.iloc[:,0].values

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

#instancier l'algo – sortie de la boucle
lda = LinearDiscriminantAnalysis()

#itérer
for i in range(n):
    #modifier l'indicateur pour i pour l'exclure de l'apprentissage
    index[i] = False
    #modéliser
    lda.fit(X=Xdata[index,:], y=Ydata[index])
    #prédire
    p = lda.predict(X=Xdata[[i],:])
    #ajouter dans le vecteur des résultats par accès indicé
    prediction[i] = p[0]
    #remettre la bonne valeur pour l'individu n°i
    index[i] = True

#afficher le taux de bonnes prédicions
from sklearn.metrics import accuracy_score
print('Acc. score = ' + str(accuracy_score(wave.onde, prediction)))

#fin
fin = time.time()

#durée
print('Duree = '+str(fin-debut)+ ' sec.')
```

Hé bien… on n'a pas gagné grand-chose finalement. Le temps de traitement moyen est resté autour de **4.6 sec**. C'est assez étonnant je trouve. Les créations de vecteurs d'index et les allocations mémoires répétées ne sont pas si pénalisantes que cela il faut croire.

En tous les cas, le profileur nous montre que split() et append() ont bel et bien disparu de la liste des instructions.



Fonction/Module	Durée totale	Appels	Durée locale
fit	4.64 sec	1000	18.94 ms
_find_and_load	1.23 sec	1479	7.65 ms
_handle_fromlist	494.15 ms	5227	2.96 ms
predict	69.49 ms	1000	2.46 ms
parser_f	14.68 ms	1	6.29 us
accuracy_score	1.09 ms	1	28.80 us
__getitem__	903.87 us	2	7.61 us
__getattr__	261.89 us	13	26.16 us
full	79.46 us	2	2.65 us
<built-in method nt.chdir>	49.66 us	1	49.66 us
values	31.45 us	1	2.32 us
<built-in method builtins.print>	19.53 us	2	11.59 us
shape	8.61 us	2	2.65 us
values	7.28 us	2	1.99 us
<built-in method time.time>	4.97 us	3	4.97 us
__init__	2.32 us	1	2.32 us

7 Conclusion

Vérifier et optimiser son code font partie du quotidien du programmeur. Dans ce tutoriel, nous avons étudié comment faire dans l'environnement de développement SPYDER pour Python. Nous constatons que nous disposons d'un bel outil qui nous permet de travailler dans d'excellentes conditions, c'est une très bonne chose.

8 Références

Spyder : The Scientific Python Development Environnement -- https://docs.spyder-ide.org/