1 Objectif

Découverte de plusieurs librairies (scikit-learn, keras / tensorflow, h2o) de Deep Learning pour Python. Implémentation de perceptrons simples et multicouches dans un problème de classement (apprentissage supervisé).

Ce document fait suite au support de cours consacré aux « Perceptrons simples et multicouches » et au tutoriel sur les « Packages R pour le Deep Learning - Perceptrons ». L'objectif est de montrer un processus complet d'analyse prédictive à l'aide de successions de commandes simples sous Python : importer les données, les préparer ; construire et configurer le réseau ; estimer ses coefficients (poids synaptiques) à partir d'un ensemble de données étiquetées ; prédire sur un second échantillon, soit aux fins de déploiement, soit aux fins d'évaluation des performances.

La tâche est en théorie relativement aisée. Le véritable enjeu pour nous est d'identifier sans ambiguïtés, d'une part les bonnes commandes, d'autre part les paramètres idoines pour construire précisément le réseau que nous souhaitons appliquer sur les données. En pratique, ce n'est pas si évident que cela parce qu'identifier de la documentation pertinente sur le web n'est pas toujours facile. On retrouve souvent le même tutoriel avec la sempiternelle base MNIST, qui est littéralement accommodée à toutes les sauces. Pouvoir généraliser la démarche à d'autres bases devient une vraie gageure. J'espère y arriver en schématisant au mieux les étapes, et surtout en donnant au lecteur la possibilité de faire le parallèle avec la même mission réalisée sous R.

Nous étudierons les packages « scikit-learn », « keras / tensorflow » et « h2o ». Nous dirons un mot également des librairies « MXNET », « PyTorch » et « caffe ».

2 Importation et préparation des données

2.1 Spambase dataset

Nous traitons la base des spams, accessible sur le serveur UCI. Il s'agit d'identifier les courriels frauduleux (spam = yes) à partir de leurs caractéristiques (fréquences des mots, de certains caractères, nombre de caractères en majuscule, etc.).

La version de Python utilisée est…

```
#version de Python
import sys
print(sys.version)
```

```
3.6.7 |Anaconda, Inc.| (default, Dec 10 2018, 20:35:02) [MSC v.1915 64 bit (AMD64)]
```

… la 3.6.7. La précision est d'importance. Elle conditionne les versions des packages de deep learning qui seront installées par la suite. Par exemple, pour « keras », il m'a fallu downgrader Python en passant de **3.7.x** à **3.6.7**.

Nous chargeons les données.

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

#charger les données
import pandas
spam_all = pandas.read_table("spambase.txt",sep="\t",header=0,decimal=".")

#vérification
spam_all.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4601 entries, 0 to 4600
Data columns (total 57 columns):
wf_make                       4601 non-null float64
wf_address                    4601 non-null float64
wf_all                        4601 non-null float64
#Etc...
capital_run_length_average    4601 non-null float64
capital_run_length_longest    4601 non-null int64
capital_run_length_total      4601 non-null int64
spam                          4601 non-null object
status                        4601 non-null object
dtypes: float64(53), int64(2), object(2)
```

Nous disposons de 4601 observations et 57 colonnes.

2.2 Subdivision apprentissage-test

La colonne « status » permet d'identifier l'appartenance aux échantillons d'apprentissage (train) et de test (test). Nous l'utilisons pour scinder les données.

```
#données d'apprentissage
spam_train = (spam_all.loc[spam_all.status=="train",:]).iloc[:,:56]
print(spam_train.shape)
```

```
(3601, 56)
```

```
#données de test
spam_test = (spam_all.loc[spam_all.status=="test",:]).iloc[:,:56]
print(spam_test.shape)
```

```
(1000, 56)
```

2.3 Standardisation des variables

Les variables étant exprimées dans des unités différentes, nous devons les standardiser. L'outil StandardScaler de « scikit-learn » permet de traiter les données d'apprentissage, et d'utiliser les mêmes paramètres (moyennes et écarts-type calculés sur l'apprentissage) pour transformer les données de test.

Nous isolons dans une structure spécifique (XTrain) les descripteurs de l'échantillon d'apprentissage.

```
#matrice des explicatives en apprentissage
XTrain = spam_train.iloc[:,:55]

#importation de la classe pour centrage réduction
from sklearn.preprocessing import StandardScaler

#instanciation
scaler = StandardScaler(with_mean=True,with_std=True)

#calcul des paramètres de centrage-réduction
scaler.fit(XTrain)

#application à l'échantillon d'apprentissage
ZTrain = scaler.transform(XTrain)

#vérification des moyennes - toutes nulles forcément
print(ZTrain.mean(axis=0))
```

```
[ 4.43965886e-17  2.95977257e-17 -2.36781806e-17 -5.91954515e-18
  9.37261315e-17 -9.07663589e-17  6.41284058e-17  6.61015875e-17
  2.86111349e-17 -3.94636343e-18 -1.18390903e-17  5.62356789e-17
 -7.89272686e-18 -1.08524994e-17 -2.12117034e-17 -2.66379532e-17
 -3.79837480e-17  2.56513623e-17 -5.87021560e-17  1.38122720e-17
  3.00910212e-17  1.38122720e-17 -3.35440892e-17  1.47988629e-18
 -5.91954515e-18 -7.89272686e-18  3.55172709e-17  1.77586354e-17
  5.57423835e-17  4.53831795e-17 -8.38602229e-18 -2.07184080e-17
 -2.36781806e-17 -4.73563612e-17  1.97318172e-17  0.00000000e+00
  1.38122720e-17  9.86590858e-19 -1.77586354e-17  2.56513623e-17
 -2.95977257e-17  1.97318172e-17  2.95977257e-17 -2.56513623e-17
  1.97318172e-17  1.72653400e-17 -3.15709074e-17  3.74904526e-17
 -8.38602229e-18 -1.97318172e-18  9.86590858e-18 -1.57854537e-17
  1.77586354e-17  1.57854537e-17  2.31848852e-17]
```

Les moyennes des variables centrées et réduites sont forcément nulles (aux erreurs de troncature près).

```
#application à l'échantillon test
ZTest = scaler.transform(spam_test.iloc[:,:55])

#affichage des moyennes des variables transformées - pas forcément nulles
print(ZTest.mean(axis=0))
```

```
[-0.0318074   0.00049163 -0.01641632 -0.05026768 -0.00563452 -0.01685142
  0.01216322  0.00029678 -0.09521854 -0.08899949 -0.02473238  0.04242856
 -0.00624254  0.0592411  -0.02253129  0.00652737  0.00364309 -0.00069244
  0.01166872  0.00031593 -0.0559309   0.02776495 -0.02159979 -0.06320722
 -0.07120785  0.01405922 -0.03224498 -0.01461215 -0.01538603 -0.06255138
  0.00292449 -0.06584005 -0.05029154 -0.06077969 -0.00415265  0.05828345
  0.02322752 -0.06473915 -0.00315202 -0.02378808 -0.01814571  0.01825538
 -0.02892168  0.02228524 -0.03345056 -0.02640054  0.00695035 -0.02600768
 -0.01712295 -0.03221673 -0.01674946  0.01682035 -0.04283227 -0.04632221
```

```
-0.04193455]
```

Ce n'est pas le cas pour l'échantillon test puisque nous avons utilisé les moyennes de l'échantillon d'apprentissage pour centrer.

2.4 Fonction d'évaluation des performances

Enfin, nous créons une fonction ad hoc pour évaluer les performances des modèles en test. Elle renvoie la matrice de confusion, le taux d'erreur et la F-Mesure (F1-Score).

```
#definition d'une fonction d'évaluation
def eval_model(y_true,y_pred,poslabel="yes"):
    #matrice de confusion
    mc = metrics.confusion_matrix(y_true,y_pred)
    print("Confusion matrix :")
    print(mc)
    #taux d'erreur
    err = 1 - metrics.accuracy_score(y_true,y_pred)
    print("Err-rate = ",err)
    #F1-score
    f1 = metrics.f1_score(spam_test.spam,y_pred,pos_label="yes")
    print("F1-Score = ",f1)
```

3 Packages Python pour le perceptron

3.1 Note sur l'installation des packages

L'installation des packages n'est pas une sinécure, loin de là. En partie, parce qu'il y a une prolifération de documentation fantaisiste (j'essaie de rester poli) sur le net. Tout le monde dit faire comme ceci, comme cela, et on s'y perd totalement, notamment parce que, selon les versions, les commandes sont différentes. Pour ma part, j'ai travaillé à partir des outils suivants :

- Mon système est Windows 10 64 bits – Version éducation
- J'ai installé la distribution Anaconda (https://www.anaconda.com/download/).
- Ma version de Python est **3.6.7** (cf. section 2.1).
- Et j'ai utilisé le gestionnaire de paquets « conda » - via la ligne de commande DOS - pour installer les différents packages, avec l'instruction « conda install paquet » dans le répertoire « Anaconda3\scripts ».

Cette procédure a fonctionné pour tous les packages cités dans ce tutoriel, y compris ceux que je n'ai finalement pas étudié dans le détail (cf. section 3.5).

3.2 Package « scikit-learn »

Le package « scikit-learn » est très populaire auprès des data scientists (« Top 8 Python Machine Learning Libraires », Dan Clark, Octobre 2018). On le comprend aisément : il est complet, facile à

utiliser et, surtout, les prototypes des fonctions sont parfaitement standardisés. On sait que fit() permet de modéliser, qu'il s'agisse d'analyse prédictive, de transformation de variables, etc. ; que transform(), permet de projeter sur des individus supplémentaires ; etc. Cette normalisation des commandes facilite l'apprentissage de la bibliothèque. Elle nous permet surtout de mieux organiser nos programmes dans les études à grande échelle. Son succès est complètement mérité, d'autant plus que les algorithmes implémentés sont de très grande qualité souvent (performances, rapidité).

Nous utilisons la version **0.20.1** de « scikit-learn ».

```
#version
import sklearn
print(sklearn.__version__)
```

```
0.20.1
```

3.2.1 Perceptron simple

Pour implémenter un perceptron simple, nous faisons appel à la classe Perceptron. Nous passons en paramètre : random_state = 100 pour que l'expérimentation soit reproductible à l'identique ; max_iter = 1500 qui est le nombre d'epochs sur la base ; tol = None pour que l'algorithme aille bien au bout des itérations demandées. Remarque : Il existe une multitude d'autres paramètres que nous passons sous silence. Selon les données traitées (nombre de classes, nombre de descripteurs, présence de variables non-pertinentes ou fortement corrélées, etc.), ils peuvent influer significativement sur la qualité du modèle induit. Leur étude justifierait la rédaction d'un tutoriel à part entière tant ils sont nombreux (Scikit-Learn – Linear Model, Perceptron).

```
#perceptron simple
from sklearn.linear_model import Perceptron

#instanciation - tol = None pour que le nombre d'itérations atteigne bien
#max_iter = 1500
ps_sklearn = Perceptron(random_state=100, max_iter = 1500, tol = None)
```

Nous réalisons l'apprentissage en passant la matrice des descripteurs standardisés ZTrain et la colonne des étiquettes spam_train.spam.

```
#apprentissage
ps_sklearn.fit(ZTrain,spam_train.spam)
```

Et nous effectuons la prédiction sur les descripteurs de l'échantillon test ZTest.

```
#prediction
preds_sklearn = ps_sklearn.predict(ZTest)
```

Il ne nous reste plus qu'à évaluer la qualité de la prédiction.

```
#appel de la fonction
eval_model(spam_test.spam,preds_sklearn)
```

```
Confusion matrix :
```

```
[[579  30]
 [ 88 303]]
Err-rate =  0.118
F1-Score =  0.8370165745856353
```

Le **taux d'erreur** en test, sur 1000 observations, est de **11.8%**. Cette valeur nous servira de référence par la suite. A priori, le perceptron multicouche, classifieur non-linéaire, devrait faire mieux (c'est ce que nous avions constaté sous R tout du moins ; Packages R, section 4).

Sous « scikit-learn », nous avons accès à des informations supplémentaires, notamment le biais du modèle (la constante),

```
#coef du biais (constante)
print(ps_sklearn.intercept_)
```

```
[-45.]
```

Et les coefficients.

```
#poids synaptiques pour les variables
coefs = ps_sklearn.coef_[0,:]
print(coefs)
```

```
[-3.97008240e+00 -2.57884442e+00  7.92529481e-01  3.60073859e+01
  4.23980215e+00 -5.57312984e+00  1.94819001e+01  5.12483902e+00
 -2.34200990e+00 -4.13381731e+00 -2.44879402e-01 -1.28764055e+00
  1.15892329e-01 -3.57170347e+00  8.21975488e+00  1.13693489e+01
  6.87052935e+00 -6.67373604e-01 -3.22032975e+00  8.09869579e+00
  1.32793699e+00  1.43997084e+00  8.01806489e+00 -7.30625194e+00
 -3.39625020e+01 -2.16535225e+00 -1.69667634e+01 -3.22335373e+00
 -5.16973452e+00 -3.90691959e+00 -5.39117357e+00 -4.35789979e+00
 -1.42213062e+01  5.86462119e+00 -6.35142240e-01 -4.46467396e+01
 -5.91432334e+00 -5.81702513e+00 -1.67107128e+02 -2.94565948e+01
 -7.95930580e+00 -1.07071191e+01 -9.04311957e+00 -1.44792720e+01
 -9.42191225e-02 -1.46124845e+01 -1.38079154e+01  9.03826035e-01
 -1.64458914e+01 -3.60390385e+00  2.56044650e+01  1.01525328e+01
  1.82606048e+00  1.52674301e+01  1.25107842e+01]
```

Les variables étant standardisées, exprimées donc sur la même échelle, il est possible de mesurer leur importance dans le modèle en les triant selon la valeur absolue décroissante des coefficients. Ainsi, les 10 variables les plus influentes sont :

```
#ordre des variables selon la valeur absolue des coefs
import numpy
index = numpy.argsort(numpy.abs(coefs))

#inversion des indices (pour tri décroissant)
index = index[::-1]

#affichage des 10 variables les plus influentes
temp = {"variable":XTrain.columns[index],"coef":coefs[index]}
print(pandas.DataFrame.from_dict(temp).head(10))
```

```
                   variable       coef
```

```
0                       wf_cs -167.107128
1                    wf_parts  -44.646740
2                       wf_3d   36.007386
3                       wf_hp  -33.962502
4                  wf_meeting  -29.456595
5                   cf_dollar   25.604465
6                   wf_remove   19.481900
7                      wf_lab  -16.966763
8                cf_sqbracket  -16.445891
9  capital_run_length_longest   15.267430
```

3.2.2 Perceptron multicouche

Nous avons des paramètres supplémentaires pour le perceptron multicouche (MLPClassifier) : nombre de couches cachées et nombre de neurones dans les couches cachées (hidden_layer_sizes, une seule couche cachée avec 2 neurones ici), la fonction d'activation (activation, sigmoïde).

```
#perceptron multicouche
from sklearn.neural_network import MLPClassifier

#instanciation
pmc_sklearn = MLPClassifier(hidden_layer_sizes=(2),activation='logistic',random_state=100,max_iter=1500)
```

Les étapes suivantes sont les mêmes que pour le perceptron simple.

```
#apprentissage
pmc_sklearn.fit(ZTrain,spam_train.spam)

#prediction
predm_sklearn = pmc_sklearn.predict(ZTest)

#evaluation
eval_model(spam_test.spam,predm_sklearn)
```

```
Confusion matrix :
[[581  28]
 [ 49 342]]
Err-rate =  0.07699999999999996
F1-Score =  0.8988173455978974
```

Le taux d'erreur en test est 7.69%. Le gap par rapport au perceptron simple (11.8%) est significatif.

De nouveau, nous pouvons afficher les coefficients. Mais leur structure est plus complexe :

```
#coefficients
print(pmc_sklearn.coefs_)
```

```
[array([[ 0.04584088,  0.11806302],
       [-0.34920056, -0.14510976],
       [-0.10217344,  0.00248883],
       [ 0.87635692,  1.04094411],
       [ 0.06469381,  1.12261441],
       [ 0.41290777, -0.04851534],
       [ 1.87639654,  1.15954378],
       [ 0.62314685,  0.0615383 ],
```

```
       [-0.11104973,  0.39505964],
       [ 0.28207276, -0.06624127],
       [-0.51462083,  0.87081215],
       [-0.08468871, -0.35269796],
       [-0.05054871,  0.01358578],
       [ 0.08273355, -0.10385426],
       [ 0.55017209, -0.14495623],
       [ 0.62442901,  1.16172904],
       [ 0.63903948,  0.15113186],
       [-0.2538749 ,  0.61249114],
       [ 0.18168258, -0.0745809 ],
       [ 0.16228321,  0.10530078],
       [ 0.1927543 ,  0.47415605],
       [ 0.2617876 ,  0.29011005],
       [ 1.04122862,  1.24890891],
       [ 0.77517705,  0.80326223],
       [-1.96893115, -2.09977323],
       [-0.82399901, -1.03443433],
       [-0.68066965, -0.70897115],
       [ 0.26111104, -0.18941414],
       [-1.0389818 , -0.84373993],
       [-0.07322674, -0.02566129],
       [-0.88811988, -0.20500045],
       [-0.29541955, -0.60386884],
       [-0.89055659, -0.9742008 ],
       [ 0.49215857,  0.45070061],
       [-0.87715026, -1.00077177],
       [ 0.07132271, -0.30454413],
       [-0.12452077, -0.31498937],
       [-0.28039867, -0.19070823],
       [-1.09514861, -0.97595841],
       [-0.94166079, -1.16533166],
       [-0.452148  , -0.37818079],
       [-0.87221425, -1.0966813 ],
       [-0.97804979, -0.96723526],
       [-1.98180958, -1.63482809],
       [-0.09840403, -0.19583323],
       [-0.73798974, -0.81464829],
       [-0.30434084, -0.29555799],
       [ 0.05504654, -0.31946823],
       [-0.26785171, -0.22837373],
       [ 1.69484537,  1.6735467 ],
       [ 1.47852872,  1.80951969],
       [ 0.34869136,  0.24795396],
       [ 1.56397622,  1.52591952],
       [ 1.33156042,  1.71434411],
       [ 1.06299193,  0.07374195]]), array([[3.53535159],
       [3.38863517]])]
```

Nous avons les connexions entre la couche d'entrée et les deux neurones de la couche cachée (en bleu), puis les connexions entre ces derniers et le neurone de la couche de sortie (en vert).

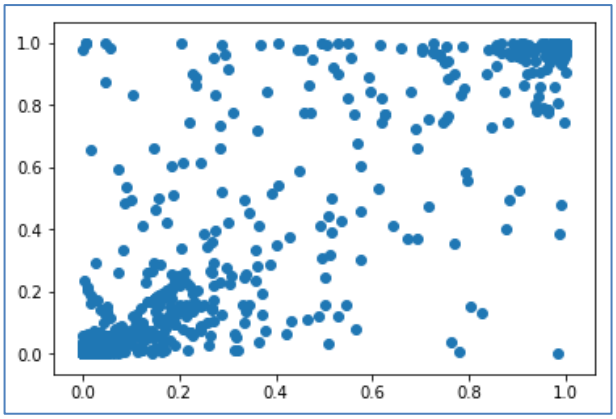
Nous pouvons aussi afficher les constantes :

```
#biais
print(pmc_sklearn.intercepts_)
```

```
[array([-0.86357942, -0.78081372]), array([-3.64458235])]
```

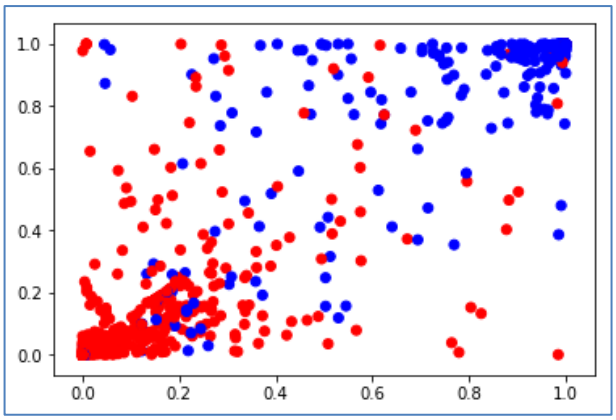
A partir de ces informations, nous pouvons calculer les coordonnées des individus de l'échantillon test dans l'espace de représentation à 2 dimensions (puisque 2 neurones) définie par la couche cachée. Nous les projetons dans le plan.

```
#calcul des coordonnées des individus dans le plan défini par la couche cachée
#combinaison linéaire
Z1 = numpy.zeros(ZTest.shape[0])
for i in range(ZTest.shape[0]):
    Z1[i] = numpy.sum(ZTest[i,:]*pmc_sklearn.coefs_[0][:,0]) + pmc_sklearn.intercepts_[0][0]
#transformation avec la fonction sigmoïde
Z1 = 1.0/(1.0+numpy.exp(-Z1))
print(Z1)

#combinaison linéaire
Z2 = numpy.zeros(ZTest.shape[0])
for i in range(ZTest.shape[0]):
    Z2[i] = numpy.sum(ZTest[i,:]*pmc_sklearn.coefs_[0][:,1]) + pmc_sklearn.intercepts_[0][1]
#transformation avec la fonction sigmoïde
Z2 = 1.0/(1.0+numpy.exp(-Z2))
print(Z2)

#matplotlib
import matplotlib.pyplot as plt

#scatter plot
plt.scatter(Z1,Z2)
```



Ce graphique n'a vraiment d'intérêt que si les observations sont étiquetées.

```
#avec des couleurs pour les classes d'appartenance
plt.scatter(Z1,Z2,c=pandas.Series(['red','blue'])[(spam_test.spam=="yes").astype('int')])
```



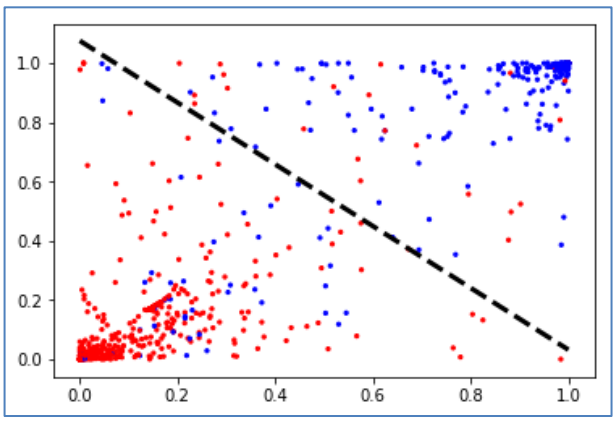
Les coefficients entre la couche de sortie et la couche cachée définissent la droite de séparation permettant de discerner les classes. Nous l'ajoutons dans le graphique.

```
#coordonnées de la frontière de séparation
horiz = numpy.linspace(0,1)
vertic = -pmc_sklearn.coefs_[1][0]/pmc_sklearn.coefs_[1][1] * horiz - pmc_sklearn.intercepts_[1]/pmc_sklearn.coefs_[1][1]

#graphique avec la frontière séparation
plt.scatter(Z1,Z2,c=pandas.Series(['red','blue'])[(spam_test.spam=="yes").astype('int')],s=5)
plt.plot(horiz,vertic,c="black",linewidth=3,linestyle="--")
plt.show()
```

Les individus mal classés sont :

- Les points rouges situés au-delà de la frontière (coin nord-est) ;
- Et les points bleus situés en deçà (coin sud-ouest).

Leur proportion est relativement faible (7.69%).



3.3 Package « keras »

Nous avons étudié « keras » dans un précédent tutoriel (« Deep Learning avec Tensorflow et Keras (Python) », avril 2018. Pour rappel, « keras » est une surcouche qui permet d'accéder

« facilement » aux fonctionnalités de la librairie « tensorflow » (d'autres moteurs sous-jacents sont possibles).

A l'importation de « keras », la console IPython nous indique le « moteur » utilisé justement.

```
#importation
import keras
```

```
Using TensorFlow backend.
```

Nous procédons à la vérification de version.

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

```
2.2.4
```

Pour définir l'architecture du réseau, nous créons une structure séquentielle (Sequential) dans laquelle nous ajoutons 2 couches Dense (entrée vers cachée + cachées vers sortie). « Dense » parce que les neurones de la couche suivante sont connectés avec tous ceux de la couche précédente.

```
#importation des classes
from keras.models import Sequential
from keras.layers import Dense

#instanciation de la structure
pmc_keras = Sequential()

#architeture du réseau
#entree vers couche cachée
pmc_keras.add(Dense(units=2,input_dim=55,activation="sigmoid"))
#couche cachée vers sortie
pmc_keras.add(Dense(units=1,activation="sigmoid"))
```

De l'entrée vers la couche cachée, nous avons 55 variables prédictives (input_dim=55) et deux neurones (units=2), « sigmoid » est la fonction d'activation. Vers la couche de sortie, nous avons un seul neurone (units = 1), suffisant pour un problème binaire, avec encore une fois la fonction d'activation sigmoïde.

Il nous faut ensuite configurer l'algorithme d'apprentissage : la fonction de perte à optimiser (loss), le moteur d'optimisation (optimizer), et la mesure de performance utilisée pour le suivi (metrics).

```
#configuration de l'apprentissage
pmc_keras.compile(loss="binary_crossentropy",optimizer="adam",metrics=["accuracy"])
```

Nous pouvons enfin lancer le processus sur l'échantillon d'apprentissage où les descripteurs sont standardisés, la variable cible recodée en 1/0 (yes/no), le nombre d'itérations sur la base complète est 150 (epochs), et la mise à jour des coefficients est réalisée lors du passage de 15 observations (bach_size).

```
#processus d'apprentissage
pmc_keras.fit(ZTrain,(spam_train.spam=="yes").astype("float"),epochs=150,batch_size=15)
```

Nous disposons d'un suivi du processus dans la console.

```
Epoch 1/150
3601/3601 [==============================] - 1s 219us/step - loss: 0.6367 - acc: 0.6332
Epoch 2/150
3601/3601 [==============================] - 0s 44us/step - loss: 0.5237 - acc: 0.8225
Epoch 3/150
3601/3601 [==============================] - 0s 42us/step - loss: 0.4515 - acc: 0.8711
Etc.
Epoch 148/150
3601/3601 [==============================] - 0s 39us/step - loss: 0.1506 - acc: 0.9481
Epoch 149/150
3601/3601 [==============================] - 0s 39us/step - loss: 0.1505 - acc: 0.9486
Epoch 150/150
3601/3601 [==============================] - 0s 42us/step - loss: 0.1504 - acc: 0.9483
```

Le taux de reconnaissance, mesuré sur l'échantillon d'apprentissage, est de 94.83% (taux d'erreur = 5.62% en resubstitution) à l'issue de la modélisation. Reste à savoir ce que le modèle donne sur l'échantillon test. Nous faisons appel à la fonction predict().

```
#prediction
proba_predm_keras = pmc_keras.predict(ZTest)

#dimension
print(proba_predm_keras.shape)  #(1000,1)

#10 premières valeurs
print(proba_predm_keras[:10,:])
```

Qui fournit une matrice à 1000 lignes (1000 individus de l'échantillon test) et 1 colonne (mais il s'agit bien d'une matrice comme nous l'indique l'instruction .shape).

Voici ce que nous avons lors de l'affichage des 10 premières valeurs.

```
[[0.9819412 ]
 [0.9825424 ]
 [0.0018578 ]
 [0.7537897 ]
 [0.96073914]
 [0.00531852]
 [0.9814042 ]
 [0.9213397 ]
 [0.00245889]
 [0.06371114]]
```

predict() produit les probabilités d'affectation aux classes en réalité.

Il faut les transformer en classes prédites en les comparant à la valeur seuil **0.5**.

```
#traduction en classement
predm_keras = numpy.repeat("yes",ZTest.shape[0])
```

```
predm_keras[proba_predm_keras[:,0] < 0.5] = "no"
```

Nous pouvons passer à l'évaluation.

```
#evaluation
eval_model(spam_test.spam,predm_keras)
```

```
Confusion matrix :
[[584  25]
 [ 60 331]]
Err-rate =  0.08499999999999996
F1-Score =  0.8862115127175367
```

Le taux d'erreur est de **8.49%**.

Avec « keras » également, nous pouvons afficher les poids synaptiques.

```
#poids
print(pmc_keras.get_weights())
```

La structure produite est décrite dans un précédent tutoriel (Keras – Python, avril 2018 ; page 10).

3.4 Package « H2o »

« H2O » est une plate-forme machine learning JAVA qui propose des API pour plusieurs langages de programmation, dont Python.

Après avoir installé la librairie, nous la chargeons :

```
#importation
import h2o
print(h2o.__version__)
```

```
3.18.0.2
```

Nous la démarrons ensuite pour accéder aux algorithmes de machine learning. Une condition nécessaire au bon fonctionnement de la librairie est la présence d'une JVM (Java Virtual Machine) en bon état de marche sur notre système.

```
#démarrage
h2o.init()
```

```
Checking whether there is an H2O instance running at http://localhost:54321..... not found.
Attempting to start a local H2O server...
; OpenJDK 64-Bit Server VM (build 25.152-b12, mixed mode)56-b12)
  Starting server from D:\Logiciels\Anaconda3\h2o_jar\h2o.jar
  Ice root: C:\Users\Zatovo\AppData\Local\Temp\tmphlsvgOvt
  JVM stdout: C:\Users\Zatovo\AppData\Local\Temp\tmphlsvgOvt\h2o_Zatovo_started_from_python.out
  JVM stderr: C:\Users\Zatovo\AppData\Local\Temp\tmphlsvgOvt\h2o_Zatovo_started_from_python.err
  Server is running at http://127.0.0.1:54321
Connecting to H2O server at http://127.0.0.1:54321... successful.
Warning: Your H2O cluster version is too old (9 months and 15 days)! Please download and
install the latest version from http://h2o.ai/download/
--------------------------  ------------------------------
H2O cluster uptime:         03 secs
H2O cluster timezone:       Europe/Paris
H2O data parsing timezone:  UTC
```

```
H2O cluster version:        3.18.0.2
H2O cluster version age:    9 months and 15 days !!!
H2O cluster name:           H2O_from_python_Zatovo_uwuxnj
H2O cluster total nodes:    1
H2O cluster free memory:    1.759 Gb
H2O cluster total cores:    8
H2O cluster allowed cores:  8
H2O cluster status:         accepting new members, healthy
H2O connection url:         http://127.0.0.1:54321
H2O connection proxy:
H2O internal security:      False
H2O API Extensions:         Algos, AutoML, Core V3, Core V4
Python version:             3.6.7 final
--------------------------  ------------------------------
```

Note : Apparemment, ma version de H20 est ancienne (9 mois et 15 jours), mais c'est celle qui a été installée automatiquement par « conda » (gestionnaire de paquets pour Anaconda), et je ne vais pas me lancer dans des manipulations non maîtrisées à ce stade. Il n'en reste pas moins que la librairie me semble particulièrement riche. Nous n'en explorons qu'une infime partie dans ce tutoriel. Je prévois de l'étudier de manière autrement plus approfondie dans un futur très proche.

Nous devons typer notre data frame Pandas au format H2O dans un premier temps.

```
#type le data frame en H2o compatible
h2oTrain = h2o.H2OFrame(spam_train)
```

Nous instancions ensuite notre perceptron multicouche après avoir importé la classe adéquate.

```
#classe deeplearning
from h2o.estimators import H2ODeepLearningEstimator

#instancier
pmc_h2o = H2ODeepLearningEstimator(standardize=True,epochs=1250,hidden=[2],seed=100,activation="Tanh",distribution="bernoulli")
```

Plusieurs commentaires : nous lui passons l'ensemble de données non-standardisée en apprentissage parce que l'outil sait le faire automatiquement (standardize = True) ; l'algorithme passera 1250 fois sur l'ensemble d'apprentissage (epochs = 1250) ; nous avons une seule couche cachée avec 2 neurones (hidden = [2]) ; nous fixons seed à 100 pour rendre l'expérimentation reproductible ; nous utilisons la fonction d'activation « tangente hyperbolique » (activation = ''Tanh'') puisque la sigmoïde n'est pas disponible ; et nous utilisons la loi de Bernoulli pour modéliser la variable cible qui est binaire (distribution = ''bernoulli'').

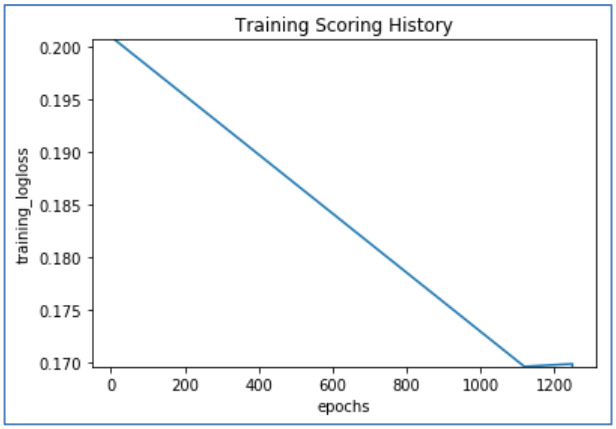
Nous pouvons lancer les calculs sur l'ensemble d'apprentissage, nous passons en paramètre : la liste des noms des variables explicatives, le nom de la variable cible, et le dataset d'apprentissage.

```
#apprentissage
pmc_h2o.train(h2oTrain.columns[:55],"spam",training_frame=h2oTrain)
```

Nous pouvons afficher la décroissance de la fonction de perte en fonction du nombre d'epochs.

```
#évolution de l'apprentissage
```

```
pmc_h2o.plot()
```



Nos devons également typer l'ensemble de test avant de pouvoir réaliser la prédiction. Le résultat doit être transformé en un data frame Pandas pour pouvoir être exploité par la suite.

```
#transformation de data test en compatible H2o
h2oTest = h2o.H2OFrame(spam_test)

#prédiction sur l'échantillon test - format H2o obtenu
predm_h2o = pmc_h2o.predict(h2oTest)

#retro-convertir en Pandas
ppredm_h2o = predm_h2o.as_data_frame()
print(ppredm_h2o.head())
```

```
  predict        no       yes
0     yes  0.042255  0.957745
1     yes  0.042255  0.957745
2      no  0.989569  0.010431
3     yes  0.042495  0.957505
4     yes  0.042736  0.957264
```

Nous disposons de 3 colonnes : la prédiction et les probabilités d'appartenance aux classes (no, yes). Nous confrontons les classes prédites avec les classes observées sur l'échantillon test.

```
#evaluation
eval_model(spam_test.spam,ppredm_h2o.predict)
```

```
Confusion matrix :
[[593  16]
 [ 67 324]]
Err-rate =  0.08299999999999996
F1-Score =  0.8864569083447332
```

Le taux d'erreur en test est **8.29%**.

« H2O » propose des fonctionnalités supplémentaires qui éveillent notre intérêt.

summary() résume l'architecture du réseau.

```
#résumé du modèle
```

```
pmc_h2o.summary()
```

```
Status of Neuron Layers: predicting spam, 2-class classification, bernoulli distribution, CrossEntropy loss, 118 weights/biases, 10,2 KB, 4Â 501Â 250 training samples, mini-batch size 1

  layer  units  type     dropout  l1  l2  mean_rate            rate_rms             momentum  mean_weight          weight_rms          mean_bias           bias_rms
--- ------- ------- -------- --------- ---- ---- -------------------- -------------------- ---------- -------------------- ------------------- ------------------- -------------------
  1      55  Input    0.0
  2       2  Tanh     0.0       0.0  0.0  0.013895847616632553 0.015586350113153458 0.0        -0.36769084019417114 2.783191680908203  0.3761290875495188  1.8021025657653809
  3       2  Softmax            0.0  0.0  0.00166085
```

Nous observons qu'il y a 55 neurones dans la couche d'entrée parce que 55 variables explicatives ; 2 neurones dans la couche intermédiaire, avec une fonction d'activation « tangente hyperbolique », c'est ce que nous avions demandé ; 2 neurones dans couche de sortie parce que la variable cible est à 2 classes, « h2o » choisit automatiquement la fonction softmax pour produire les probabilités conditionnelles. Il y avait 118 coefficients à estimer en comptabilisant les biais [(55 + 1) x 2 + (2 + 1) x 2 = 118].

Avec show(), nous disposons d'un affichage plus détaillé des résultats.

```
#affichage
pmc_h2o.show()
```

```
Model Details
=============
H2ODeepLearningEstimator :  Deep Learning
Model Key:  DeepLearning_model_python_1545656231910_1


ModelMetricsBinomial: deeplearning
** Reported on train data. **

MSE: 0.04534617208784071
RMSE: 0.21294640660936429
LogLoss: 0.16956387052275546
Mean Per-Class Error: 0.0622316072935043
AUC: 0.9736428922285285
Gini: 0.947285784457057
Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.572470248874215:
       no    yes    Error    Rate
-----  ----  -----  -------  --------------
no     2113  66     0.0303   (66.0/2179.0)
yes    135   1287   0.0949   (135.0/1422.0)
Total  2248  1353   0.0558   (201.0/3601.0)
Maximum Metrics: Maximum metrics at their respective thresholds

metric                       threshold    value     idx
---------------------------  -----------  --------  -----
max f1                       0.57247      0.927568  101
max f2                       0.190852     0.944399  252
max f0point5                 0.892101     0.943624  55
max accuracy                 0.594527     0.944182  99
max precision                0.957583     0.962036  2
max recall                   0.0104314    1         399
max specificity              0.957744     0.980266  0
max absolute_mcc             0.594527     0.882946  99
max min_per_class_accuracy   0.378762     0.933915  166
max mean_per_class_accuracy  0.475231     0.937768  112
Gains/Lift Table: Avg response rate: 39,49 %
```

	group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	cumulative_response_rate	capture_rate	cumulative_capture_rate	gain	cumulative_gain
	1	0.152458	0.957745	2.45855	2.45855	0.970856	0.970856	0.374824	0.374824	145.855	145.855
	2	0.200222	0.957745	2.45873	2.45859	0.97093	0.970874	0.11744	0.492264	145.873	145.859
	3	0.300194	0.957724	2.39166	2.4363	0.944444	0.962072	0.2391	0.731364	139.166	143.63
	4	0.400167	0.415436	1.94147	2.31268	0.766667	0.913255	0.194093	0.925457	94.1467	131.268
	5	0.500139	0.0252549	0.597916	1.96992	0.236111	0.777901	0.059775	0.985232	-40.2084	96.9917
	6	0.600111	0.0109543	0.0281372	1.64644	0.0111111	0.650162	0.00281294	0.988045	-97.1863	64.6437
	7	0.700083	0.0104361	0.0492401	1.41836	0.0194444	0.560095	0.00492264	0.992968	-95.076	41.8356
	8	0.800056	0.0104306	0.0351715	1.24552	0.0138889	0.491843	0.00351617	0.996484	-96.4828	24.5518
	9	0.900305	0.0104306	0.0280593	1.10995	0.0110803	0.43831	0.00281294	0.999297	-97.1941	10.9953
	10	1	0.0104306	0.0070539	1	0.00278552	0.39489	0.000703235	1	-99.2946	0

Scoring History:

	timestamp	duration	training_speed	epochs	iterations	samples	training_rmse	training_logloss	training_auc	training_lift	training_classification_error
	2018-12-24 14:22:34	0.000 sec		0	0	0	nan	nan	nan	nan	nan
	2018-12-24 14:22:34	0.470 sec	279147 obs/sec	10	1	36010	0.236409	0.200621	0.971817	2.32702	0.0694252
	2018-12-24 14:22:39	5.370 sec	808888 obs/sec	1120	112	4.03312e+06	0.212946	0.169564	0.973643	2.45855	0.0558178
	2018-12-24 14:22:40	5.898 sec	820348 obs/sec	1250	125	4.50125e+06	0.213086	0.169819	0.972753	2.45732	0.0558178
	2018-12-24 14:22:40	5.918 sec	820049 obs/sec	1250	125	4.50125e+06	0.212946	0.169564	0.973643	2.45855	0.0558178

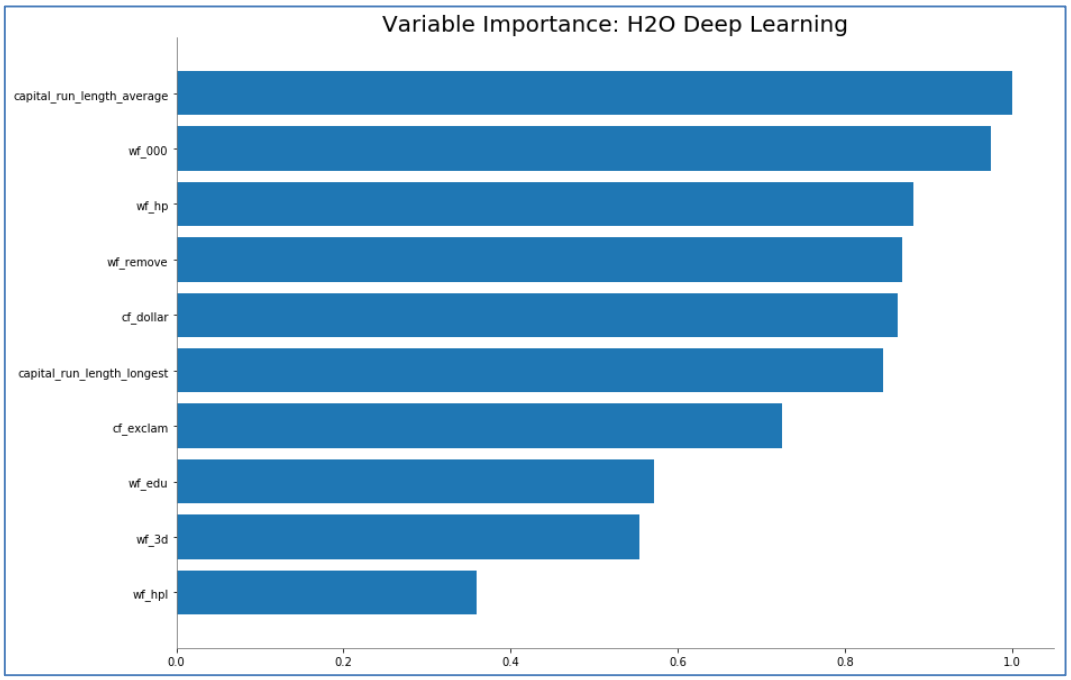
Variable Importances:

variable	relative_importance	scaled_importance	percentage
capital_run_length_average	1.0	1.0	0.06284726026788662
wf_000	0.9751128554344177	0.9751128554344177	0.06128317141604894
wf_hp	0.881908655166626	0.881908655166626	0.05542554278375881
wf_remove	0.8689811825752258	0.8689811825752258	0.05461308654920111
cf_dollar	0.8640486598014832	0.8640486598014832	0.05430309100666243
---	---	---	---
wf_original	0.04922392591834068	0.04922392591834068	0.0030935888835971264
wf_people	0.0466492660343647	0.0466492660343647	0.002931778563767601
wf_parts	0.042827218770980835	0.042827218770980835	0.0026915733646495515
wf_table	0.029552336782217026	0.029552336782217026	0.0018572834012762322
wf_make	0.015273143537342548	0.015273143537342548	0.0009598752270001575

« H2O » est prolixe. Le dernier tableau attire notre attention. La librairie propose une mesure de l'influence des variables du modèle. Dans un classifieur linéaire, elle est relativement facile à calculer si les variables sont standardisées. Mais nous avons un classifieur non-linéaire ici, l'affaire est moins évidente. J'avais fait le même commentaire sous R. En inspectant la documentation en ligne, j'ai lu le descriptif suivant : « Variable importances for Neural Network models are notoriously difficult to compute, and there are many pitfalls. H2O Deep Learning has implemented the method of Gedeon, and returns relative variable importances in descending order of importance. ». Bien évidemment que l'on va regarder tout cela dans le détail très prochainement.

Pour l'heure, on se contente d'utiliser les outils qui peuvent être graphiques :

```
#importance des variables - graphique
pmc_h2o.varimp_plot()
```



Ou, plus prosaïquement, produisent le tableau des contributions.

	0	1	2	3
0	capital_run_length_average	1.000000	1.000000	0.062847
1	wf_000	0.975113	0.975113	0.061283
2	wf_hp	0.881909	0.881909	0.055426
3	wf_remove	0.868981	0.868981	0.054613
4	cf_dollar	0.864049	0.864049	0.054303
5	capital_run_length_longest	0.845869	0.845869	0.053161
6	cf_exclam	0.725199	0.725199	0.045577
7	wf_edu	0.572352	0.572352	0.035971
8	wf_3d	0.554984	0.554984	0.034879
9	wf_hpl	0.359907	0.359907	0.022619
10	wf_free	0.350745	0.350745	0.022043
Etc.				
52	wf_parts	0.042827	0.042827	0.002692
53	wf_table	0.029552	0.029552	0.001857

```
54                      wf_make  0.015273  0.015273  0.000960
```

Enfin, il nous est possible d'obtenir un rapport détaillé sur les performances en test.

```
#performances sur l'échantillon test - analyse détaillée
pmc_h2o.model_performance(h2oTest)
ModelMetricsBinomial: deeplearning
** Reported on test data. **

MSE: 0.06943406620027617
RMSE: 0.2635034462777976
LogLoss: 0.2748085529600903
Mean Per-Class Error: 0.09204221418702419
AUC: 0.9534833423624323
Gini: 0.9069666847248645

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.4210540398184732:
       no    yes    Error    Rate
-----  ----  -----  -------  -------------
no     585   24     0.0394   (24.0/609.0)
yes    57    334    0.1458   (57.0/391.0)
Total  642   358    0.081    (81.0/1000.0)

Maximum Metrics: Maximum metrics at their respective thresholds

metric                        threshold    value     idx
----------------------------  -----------  --------  -----
max f1                        0.421054     0.891856  99
max f2                        0.247273     0.893939  149
max f0point5                  0.694769     0.927954  77
max accuracy                  0.563615     0.919     83
max precision                 0.957745     0.971193  1
max recall                    0.0104306    1         399
max specificity               0.957745     0.988506  0
max absolute_mcc              0.563615     0.830628  83
max min_per_class_accuracy    0.247273     0.898194  149
max mean_per_class_accuracy   0.417123     0.907958  106
Gains/Lift Table: Avg response rate: 39,10 %
```

group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	cumulative_response_rate	capture_rate	cumulative_capture_rate	gain	cumulative_gain
1	0.15	0.957745	2.50639	2.50639	0.98	0.98	0.375959	0.375959	150.639	150.639
2	0.155	0.957745	2.55754	2.50804	1	0.980645	0.0127877	0.388747	155.754	150.804
3	0.2	0.957745	2.38704	2.48082	0.933333	0.97	0.107417	0.496164	138.704	148.082
4	0.3	0.957487	2.40409	2.45524	0.94	0.96	0.240409	0.736573	140.409	145.524
5	0.401	0.248687	1.46869	2.20676	0.574257	0.862843	0.148338	0.88491	46.8689	120.676
6	0.5	0.0188456	0.568343	1.88235	0.222222	0.736	0.056266	0.941176	-43.1657	88.2353
7	0.6	0.0109006	0.230179	1.60699	0.09	0.628333	0.0230179	0.964194	-76.9821	60.6991
8	0.7	0.010435	0.204604	1.40665	0.08	0.55	0.0204604	0.984655	-79.5396	40.665
9	0.8	0.0104306	0.0255754	1.23402	0.01	0.4825	0.00255754	0.987212	-97.4425	23.4015
10	0.9	0.0104306	0.127877	1.11111	0.05	0.434444	0.0127877	1	-87.2123	11.1111
11	1	0.0104306	0	1	0	0.391	0	1	-100	0

La partie basse des résultats (en gris) montre le détail du calcul des lifts à partir de l'échantillon test décomposé en déciles. Pour les initiés peut-être, mais honnêtement je ne vois pas vraiment l'intérêt de s'y plonger.

Plus intéressant dans la partie médiane (Maximum metrics), nous disposons des valeurs des différentes mesures de performances (F1-score, taux de reconnaissance, précision, rappel, etc.) en fonction du seuil d'affectation. Ce dernier est égal à 0.5 habituellement dans un problème à 2 classes. Mais en réalité, nous pouvons le moduler en fonction de nos attentes par rapport au classifier. Le tableau montre que nous pouvons obtenir des gains substantiels en choisissant la bonne valeur en fonction du critère à optimiser. Par exemple, sur notre échantillon test, avec le seuil standard de 0.5, notre taux d'erreur est égal à 8.29% (page 15). Si nous le passons à 0.563615, il sera de (1 – accuracy = 1 – 0.919) = 8.1% (1.9% sur 1000 observations, c'est 19 individus bien classés en plus !).

Vérifions si cela est vrai puisque nous disposons des probabilités d'affectation. Reproduisons les calculs avec le nouveau seuil de 0.563615.

```
#vérification avec seuil
pp2 = numpy.repeat("yes",ZTest.shape[0])
pp2[ppredm_h2o.loc[:,"yes"] < 0.563615] = "no"
eval_model(spam_test.spam,pp2)
```

```
Confusion matrix :
[[593  16]
 [ 65 326]]
Err-rate =  0.08099999999999996
F1-Score =  0.8894952251023194
```

Le taux d'erreur est bien de 8.1% (aux erreurs d'arrondi près).

On trouve rarement ce point de vue (modulation des seuils d'affectation pour optimiser les performances) dans les outils de machine learning. Il était intéressant de pouvoir s'attarder dessus.

3.5 Autres packages – « MXNET », « PyTorch », « Caffe »

Je souhaitais étudier d'autres package au départ, notamment « MXNET », « PyTorch » et « Caffe » qui sont souvent cités dans les surveys en ligne. J'ai pu les installer sans difficultés dans un premier temps. Les difficultés ont commencé lorsque j'ai voulu appliquer notre analyse type.

Déjà la documentation est très rare. Les seuls tutoriels disponibles s'acharnent sur la base MNIST, en se copiant sans vergogne les uns les autres d'ailleurs. Et, quand enfin j'ai trouvé un document générique qui tenait à peu près la route (« Building simple artificial neural networks with TensorFlow, Keras, PyTorch and MXNet/Gluon », Sausheong, Mai 2018) (qui s'excite sur la base MNIST aussi d'ailleurs), je me suis rendu compte que ces outils étaient faiblement encapsulés. Ils nous imposent une programmation quasiment « à partir de zéro » de la descente du gradient pour l'apprentissage, puis de l'application des coefficients et des transformations pour le déploiement. Bien sûr, en copiant le code accessible en ligne et en l'adaptant à notre problème, nous devrions pouvoir obtenir les résultats attendus. Mais nous sommes loin de notre cahier des charges initial où l'objectif était de mener une analyse complète à l'aide d'une succession de commandes simples. J'ai préféré laisser ces librairies de côté finalement.

4 Conclusion

L'objectif de ce tutoriel était de montrer à l'aide d'un exemple concret la mise en œuvre des packages « deep learning » sous Python dans un problème prédictif modélisé à l'aide d'un perceptron multicouche. Il fallait réaliser l'étude via une succession de commandes simples, sans connaissances particulières en informatique (programmation). « scikit-learn », « keras / tensorflow » et « h2o » ont parfaitement répondu à notre cahier des charges.