1 Objectif

Découverte des librairies de Deep Learning Tensorflow / Keras pour Python. Implémentation de perceptrons simples et multicouches dans des problèmes de classement (apprentissage supervisé).

« Deep learning », « Tensorflow », « Keras »… ouh là là, plus racoleur tu meurs. Bon, j'en ai tellement entendu parler dernièrement, mes étudiants sont dans une telle attente par rapport à ces techniques et technologies, que je me suis décidé à rédiger une série de cours sur le sujet durant cet été. Et, comme d'habitude, quand je commence à m'intéresser à des algorithmes de machine learning, je regarde d'abord les outils disponibles. Il ne sert à rien de développer des enseignements sur des méthodes, fussent-elles extraordinaires, si elles ne sont disponibles dans aucun logiciel. Elles sont de facto inutilisables.

Je souhaitais travailler sous Python, au moins dans un premier temps (un tutoriel pour R viendra). Sur le podium des librairies récentes les plus populaires figurent Tensorflow, Sckit-learn et Keras (« Top 20 - Python AI and Machine Learning Open Source Projects », KDnuggets Polls, Février 2018). J'avais écrit un petit guide sur « scikit-learn » qui m'avait permis de cerner les réelles possibilité de Python en machine learning il y a un moment déjà (« Python - Machine Learning avec scikit-learn », Tutoriel Tanagra, Septembre 2015). Reste à explorer Tensorflow et Keras qui, ça tombe bien, sont clairement estampillés « deep learning » si l'on se réfère aux documents disponibles sur le web.

Tensorflow est une bibliothèque open-source développée par l'équipe Google Brain qui l'utilisait initialement en interne. Elle implémente des méthodes d'apprentissage automatique basées sur le principe des réseaux de neurones profonds (deep learning). Une API Python est disponible. Nous pouvons l'exploiter directement dans un programme rédigé en Python. C'est faisable, il existe des tutoriels et des ouvrages à ce sujet. Pourtant, j'ai préféré passer par Keras parce que le formalisme imposé par Tensorflow est déroutant au possible pour un néophyte. Découvrir de nouveaux algorithmes devient vite rédhibitoire si on a du mal à se dépatouiller avec un outil que nous sommes censés utiliser pour les mettre en application.

Keras est une librairie Python qui encapsule l'accès aux fonctions proposées par plusieurs librairies de machine learning, en particulier Tensorflow. **De fait, Keras n'implémente pas**

nativement les méthodes. Elle sert d'interface avec Tensorflow simplement. Mais pourquoi alors s'enquiquiner avec une surcouche supplémentaire direz-vous ? Parce qu'elle nous facilite grandement la vie en proposant des fonctions et procédures relativement simples à mettre en œuvre. Un apprenant qui a déjà assimilé les démarches types du machine learning, qui a pu par ailleurs utiliser des librairies qui font référence telles que scikit-learn, ne sera pas dépaysé lorsqu'il aura à travailler avec Keras. L'accès aux fonctionnalités de Tensorflow devenant transparentes, il pourra se focaliser sur la compréhension des méthodes.

Ce tutoriel a pour objectif la prise en main des outils. Pour aller à l'essentiel, nous implémenterons des perceptrons simples et multicouches dans des problèmes d'analyse prédictive. Ayant déjà nos repères concernant ces méthodes (RAK, avril 2013), nous pourrons nous consacrer pleinement à l'assimilation du mode de fonctionnement de Tensorflow / Keras. Les supports de cours consacrés aux méthodes de Deep Learning suivront.

2 Installation des librairies sous Python / Anaconda

Voici précisément les outils utilisés dans ce tutoriel (**10/04/2018**) :

- Windows 10 Education – FR – 64 bits ;
- Distribution Anaconda Python (Anaconda 5.1 pour Python 3.6) : https://www.anaconda.com/download/ ;
- Tensorflow pour Anaconda / Python (https://www.tensorflow.org/install/install_windows) ;
- Keras pour Anaconda / Python (https://anaconda.org/conda-forge/keras)

Il faut être très vigilant durant cette phase. Les problèmes d'installation sont vite arrivés. Il sera difficile par la suite de faire la part entre les erreurs de configuration machine et les appels de fonctions erronés.

3 Problème artificiel binaire

3.1 Données

Nous avions utilisé ce jeu de données précédemment (RAK, 2013 ; section 2). Il s'agit d'un problème de discrimination binaire dans le plan. La frontière séparant les classes prend la forme d'une parabole (Figure 1) :

$$\text{Si } (0.1 * X_2 > {X_1}^2) \text{ Alors } (Y = \text{positif}) \text{ Sinon } (Y = \text{négatif})$$

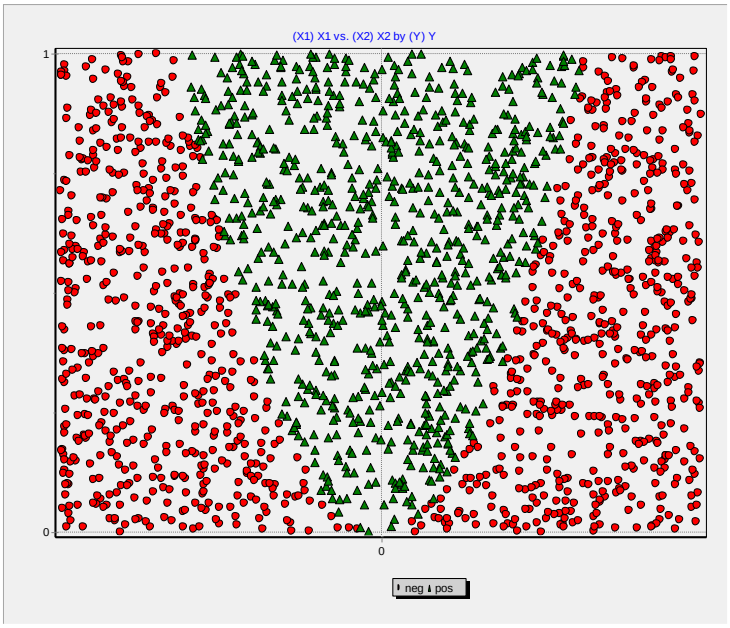

Figure 1 - Problème de discrimination binaire

3.2 Préparation des données

Importation des données. Nous importons le fichier « **artificial2d_data2.txt** » à l'aide de la fonction read_table() libraire Pandas.

```
#changement du dossier par défaut
import os
os.chdir("… votre dossier …")

#importation des données
import pandas
D = pandas.read_table("artificial2d_data2.txt",sep="\t",header=0)

#liste des variables
print(D.info())
```

Nous avons un data frame avec 2000 observations et 3 variables.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 3 columns):
X1    2000 non-null float64
X2    2000 non-null float64
Y     2000 non-null object
dtypes: float64(2), object(1)
```

Recodage de la variable cible. Notre variable Y prend ses valeurs dans {pos, neg}…

```
#décompte des classes
print(pandas.value_counts(D.Y))
```

… avec respectivement 859 et 1141 observations.

```
neg    1141
pos     859
Name: Y, dtype: int64
```

Nous devons la recoder en {1, 0} avant de pouvoir l'utiliser.

Nous créons la variable y à cet effet avec les instructions suivantes :

```
#librairie numpy
import numpy

#un vecteur de 0
y = numpy.zeros(D.shape[0])

#mettre 1 pour les "pos"
y[D.Y=="pos"] = 1

#vérification
print(numpy.sum(y))
```

La somme des valeurs est égale à 859.0. Elle correspond au nombre d'observations positives de notre jeu de données.

Subdivision en échantillons d'apprentissage et de test. Nous partitionnons les données en 1500 observations pour l'apprentissage, 500 pour le test. Nous nous assurons d'avoir les mêmes proportions d'observations positives dans les deux sous-échantillons par un tirage stratifié.

```
#isoler les descripteurs
X = D.iloc[:,:2]

#subdision 500 en test (et donc 1500 en apprentissage)
from sklearn import model_selection
XTrain,XTest,yTrain,yTest = model_selection.train_test_split(X,y,test_size=500,random_state=1,stratify=y)

#vérification
print(numpy.mean(yTrain),numpy.mean(yTest))
```

Les répartitions des classes sont respectées.

```
print(numpy.mean(yTrain),numpy.mean(yTest))
0.42933333333333334 0.43
```

A ce stade, nous sommes parés pour lancer le processus d'apprentissage supervisé.

3.3 Perceptron simple

Architecture du réseau. Nous importons les classes Sequential et Dense pour définir notre modèle et son architecture.

```
#keras
from keras.models import Sequential
from keras.layers import Dense
```

La classe Sequential est une structure, initialement vide, qui permet de définir un empilement de couches de neurones (https://keras.io/getting-started/sequential-model-guide/) :

```
#instanciation du modèle
modelSimple = Sequential()
```

Note : Sequential parce que les couches de neurones vont être ajoutées séquentiellement.

Pour spécifier un perceptron simple, nous ajoutons une couche qui relie directement la couche d'entrée (input_dim, nombre de neurones = nombre de variables prédictives) avec la couche de sortie (units = 1, une seule sortie puisque la variable cible est binaire, codée 1/0), avec une fonction d'activation sigmoïde (activation) :

```
#architecture
modelSimple.add(Dense(units=1,input_dim=2,activation="sigmoid"))#ou input_shape=(2,)
```

Note : Dense parce que tous les neurones de couche précédente seront connectés à tous les neurones de la couche suivante.

Voyons la configuration de notre modèle à ce stade :

```
#configuration de l'objet
print(modelSimple.get_config())
```

Plusieurs informations apparaissent…

```
[{'class_name': 'Dense', 'config': {'name': 'dense_1', 'trainable': True,
'batch_input_shape': (None, 2), 'dtype': 'float32', 'units': 1, 'activation':
'sigmoid', 'use_bias': True, 'kernel_initializer': {'class_name':
'VarianceScaling', 'config': {'scale': 1.0, 'mode': 'fan_avg', 'distribution':
'uniform', 'seed': None}}, 'bias_initializer': {'class_name': 'Zeros', 'config':
{}}, 'kernel_regularizer': None, 'bias_regularizer': None, 'activity_regularizer':
None, 'kernel_constraint': None, 'bias_constraint': None}}]
```

… nous retiendrons surtout que notre perceptron est pourvu du biais (ou intercept) avec l'option « use_bias : True » c.-à-d. un neurone qui prend systématiquement la valeur 1, ce qui évite à l'hyperplan séparateur de passer nécessairement par l'origine.

Visuellement, notre réseau ressemble à ceci (Figure 2) :

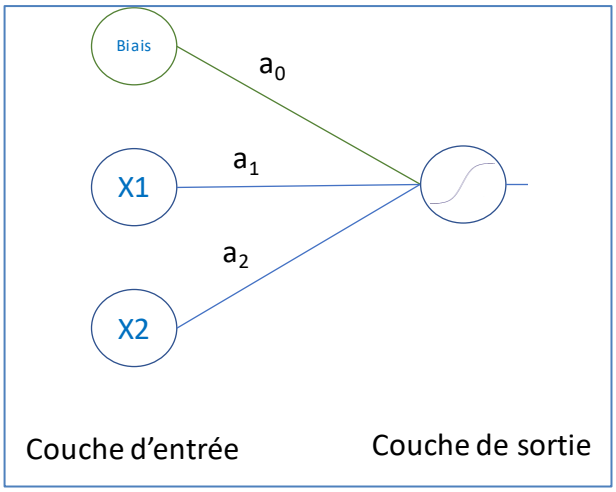

Figure 2 - Architecture de notre perceptron simple

En entrée du neurone de la couche de sortie, nous avons la combinaison linéaire :

$$d(X) = a_0 + a_1 X_1 + a_2 X_2$$

Avec l'application de la fonction d'activation sigmoïde, nous avons en sortie du neurone de la couche de sortie

$$g(d) = \frac{1}{1 + e^{-d}}$$

g(d) est une estimation de la probabilité conditionnelle P(Y = pos / X_1, X_2), déterminante dans les problèmes de classement.

Algorithme d'apprentissage. L'étape suivante consiste à spécifier les caractéristiques de l'algorithme d'apprentissage : la fonction de perte à optimiser (loss) est l'entropie croisée binaire, elle correspond à la log-vraisemblance d'un échantillon où la probabilité conditionnelle d'appartenance aux classes est modélisée à l'aide de la loi binomiale (voir R.R., « Pratique de la régression logistique », section 1.4) ; Adam est l'algorithme d'optimisation (optimizer), c'est une alternative efficace à la descente du gradient stochastique ; la métrique (metrics) utilisée pour mesurer la qualité de la modélisation est le taux de reconnaissance ou taux de succès (accuracy en anglais, à ne pas confondre avec la précision qui est un terme technique dont la formule est différente).

```
#compilation - algorithme d'apprentissage
modelSimple.compile(loss="binary_crossentropy",optimizer="adam",metrics=["accuracy"])
```

Estimation des paramètres du réseau sur l'échantillon d'apprentissage. Nous pouvons lancer l'estimation des poids synaptiques (coefficients) du réseau à partir des données étiquetées.

```
#apprentissage
modelSimple.fit(XTrain,yTrain,epochs=150,batch_size=10)
```

epochs est le nombre maximum d'itérations ; batch_size correspond aux nombre d'observations que l'on fait passer avant de remettre à jour les poids synaptiques.

L'évolution de l'apprentissage est affichée dans la console IPython. En ce qui me concerne, voici les valeurs finales de loss = 0.6374 et accuracy = 0.6387. La similitude des valeurs est une coïncidence pour notre exemple.

Une fois l'apprentissage finalisée, nous pouvons afficher les poids estimés :

```
#poids synaptiques
print(modelSimple.get_weights())
```

Nous obtenons :

```
#print(modelSimple.get_weights())
[array([[0.1563818],
       [2.0969372]], dtype=float32), array([-1.3740215], dtype=float32)]
```

Tout l'enjeu est de pouvoir les replacer dans notre structure (Figure 2) : nous avons une matrice avec deux lignes, il s'agit des poids reliant les 2 neurones de la couche d'entrée à la sortie ; le biais est la valeur dans un vecteur à part (Figure 3).

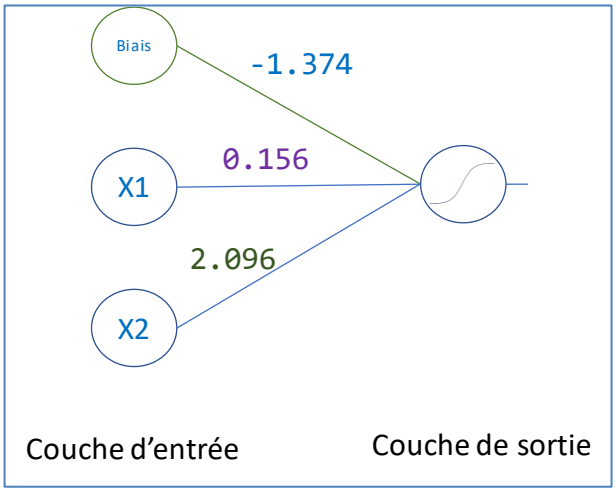

Figure 3 - Perceptron simple - Poids synaptiques estimés

Ces poids sont assez similaires à ceux que nous avions avec Tanagra dans un précédent tutoriel (RAK, 2013 ; page 5). Les structures de réseaux sont identiques, mais l'algorithme est différent : Tanagra optimise l'erreur quadratique et s'appuie sur un gradient stochastique (une approche incrémentale même, les poids sont recalculés au passage de chaque individu).

3.4 Evaluation du modèle

Prédiction et confrontation. L'approche usuelle d'évaluation consiste à réaliser la prédiction sur l'échantillon test, puis à la confronter avec les valeurs observées de la variable cible. Voici le code pour la première étape :

```
#prédiction sur l'échantillon test
predSimple = modelSimple.predict_classes(XTest)
print(predSimple[:10])
```

Nous avons des valeurs 0/1 correspondant à négatif/positif :

```
#print(predSimple[:10])
[[0]
 [1]
 [0]
 [0]
 [0]
 [1]
 [0]
 [1]
 [1]
 [0]]
```

Remarque : Il est possible d'obtenir les scores d'appartenance, pour le scoring ou encore pour l'élaboration de la courbe ROC, avec la commande predict_proba().

Nous utilisons les outils de la librairie scikit-learn pour calculer la matrice de confusion :

```
#matrice de confusion
from sklearn import metrics
print(metrics.confusion_matrix(yTest,predSimple))
```

```
[[227  58]
 [121  94]]
```

Et le taux de reconnaissance :

```
#taux de succès
print(metrics.accuracy_score(yTest,predSimple))
```

Qui est égal à **0.642**.

Fonction d'évaluation. L'autre solution consiste à utiliser l'outil dédié de la librairie Keras.

```
#outil dédié
score = modelSimple.evaluate(XTest,yTest)
print(score)
```

Il fournit la perte (loss) et le taux de reconnaissance.

```
#print(score)
[0.6342076244354248, 0.6419999990463257]
```

3.5 Perceptron multicouche

Dans cette section, nous passons à un perceptron multicouche. Nous créons toujours une structure Sequential, dans lequel nous ajoutons successivement deux objets Dense : le premier fait la jonction entre la couche d'entrée (d'où l'option input_dim indiquant le nombre de variables prédictives) et la couche cachée qui comporte (units = 3) neurones ; le second entre cette couche cachée et la sortie à un seul neurone (units = 1). Nous avons une fonction d'activation sigmoïde dans les deux cas.

```
#modélisation
modelMc = Sequential()
modelMc.add(Dense(units=3,input_dim=2,activation="sigmoid"))
modelMc.add(Dense(units=1,activation="sigmoid"))
```

Notre perceptron prend cette forme (Figure 4) :

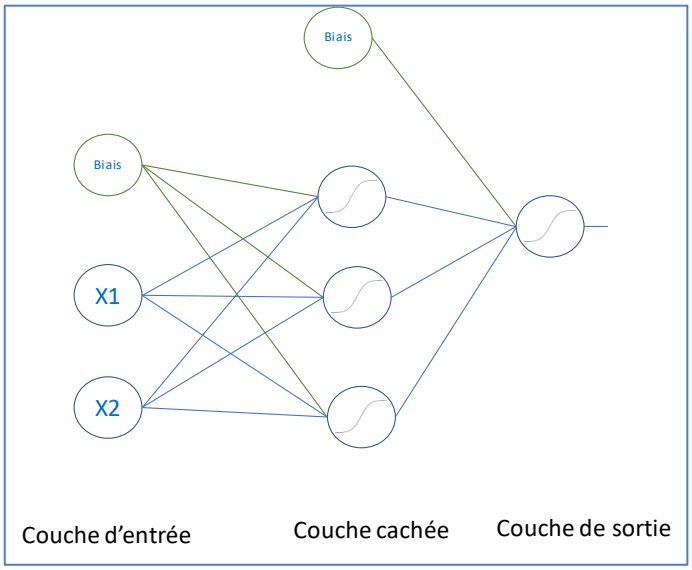

Figure 4 - Perceptron multicouche - Structure

Les étapes suivantes sont usuelles.

```
#compilation - algorithme d'apprentissage
modelMc.compile(loss="binary_crossentropy",optimizer="adam",metrics=["accuracy"])

#apprentissage
modelMc.fit(XTrain,yTrain,epochs=150,batch_size=10)

#poids synaptiques
print(modelMc.get_weights())
```

Les poids sont en adéquation avec la structure :

```
#print(modelMc.get_weights())
```

```
[array([[10.633718 , -3.2814596, 11.527278 ],
       [-1.9189957, -1.9527116,  2.2497365]], dtype=float32), array([-1.5288157 ,
0.35877246,  1.3036181 ], dtype=float32), array([[-7.19563  ],
       [-2.775277 ],
       [ 4.7566166]], dtype=float32), array([-0.7910545], dtype=float32)]
```

Nous avons une première matrice 2 lignes (parce que 2 neurones dans la couche d'entrée) et 3 colonnes (parce que 3 neurones dans la couche intermédiaire) ; un vecteur avec 3 valeurs (lien entre le biais de la couche d'entrée et les 3 neurones de la couche cachée) ; une matrice avec 3 lignes (3 neurones de la couche cachée) et 1 colonne (1 neurone de la couche de sortie) ; et enfin un vecteur avec une valeur (lien entre le biais et la couche de sortie) (Figure 5).

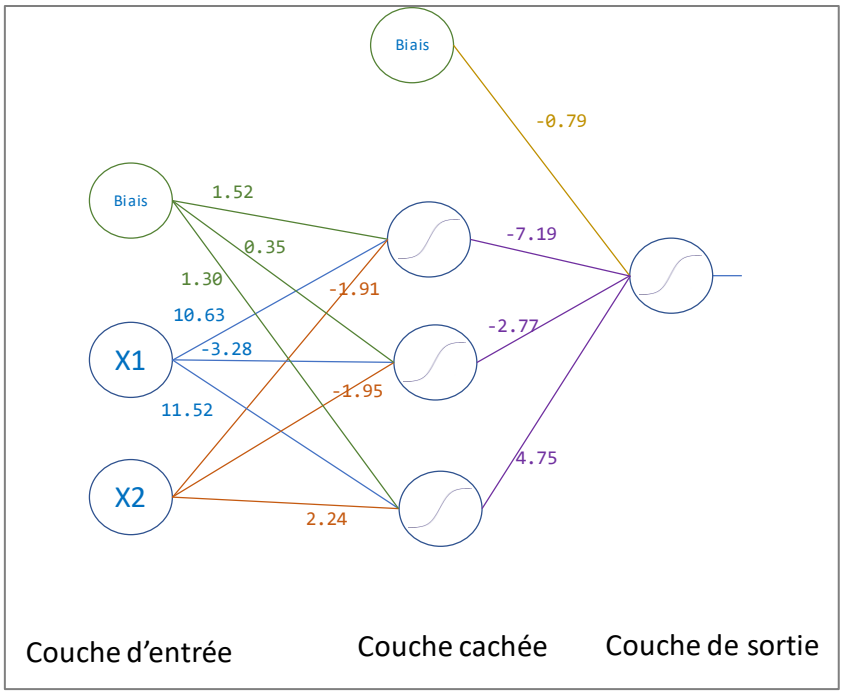

Figure 5 - Perceptron multicouche - Poids synaptiques estimés

Les performances en test…

```
#score
score = modelMc.evaluate(XTest,yTest)
print(score)
```

… correspondent à un taux de reconnaissance de **98.2%**.

4 Un exemple multi classes réaliste

Maintenant que nous avons compris le principe, nous traitons un exemple un peu plus réalise dans cette partie avec une préparation des données spécifique (centrage-réduction des variables) et une variable cible comportant 3 classes.

4.1 Données

Nous utilisons les données WINE (https://archive.ics.uci.edu/ml/datasets/wine) accessibles sur le dépôt UCI. L'objectif est de discerner 3 vignerons (cultivars) d'une même région à partir des caractéristiques des vins qu'ils produisent. Voici les premières lignes du fichier « **wine.txt** »:

```
cultivars,alcohol,malic,ash,alcalinity,magnesium,tot_phenols,flavonoids,nonf_phenols,proanth,color,hue,od280,proline
1,14.23,1.71,2.43,15.6,127,2.8,3.06,.28,2.29,5.64,1.04,3.92,1065
1,13.2,1.78,2.14,11.2,100,2.65,2.76,.26,1.28,4.38,1.05,3.4,1050
1,13.16,2.36,2.67,18.6,101,2.8,3.24,.3,2.81,5.68,1.03,3.17,1185
1,14.37,1.95,2.5,16.8,113,3.85,3.49,.24,2.18,7.8,.86,3.45,1480
1,13.24,2.59,2.87,21,118,2.8,2.69,.39,1.82,4.32,1.04,2.93,735
...
```

La variable cible « cultivars » est codée en $\{1, 2, 3\}$.

Nous importons les données et nous listons les variables.

```
#changement du dossier par défaut
import os
os.chdir("… votre dossier …")

#importation des données
import pandas
D = pandas.read_table("wine.txt",sep=",",header=0)

#liste des variables
print(D.info())
```

Nous disposons de 14 variables, la cible + 13 prédictives, et de 178 observations.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 178 entries, 0 to 177
Data columns (total 14 columns):
cultivars       178 non-null int64
alcohol         178 non-null float64
malic           178 non-null float64
ash             178 non-null float64
alcalinity      178 non-null float64
magnesium       178 non-null int64
tot_phenols     178 non-null float64
flavonoids      178 non-null float64
nonf_phenols    178 non-null float64
proanth         178 non-null float64
color           178 non-null float64
hue             178 non-null float64
od280           178 non-null float64
proline         178 non-null int64
dtypes: float64(11), int64(3)
```

4.2 Préparation des données

Recodage de la variable cible. Keras ne sait pas manipuler directement une variable cible multivaluée. Il faut la transformer en une série d'indicatrices 0/1. Dans notre cas, « cultivars » prend ses valeurs dans $\{1, 2, 3\}$, il faudra donc produire 3 variables binaires, une pour chaque classe.

Vérifions tout d'abord le nombre d'observations pour chaque classe :

```
#décompte des classes
print(pandas.value_counts(D.cultivars))
```

59 observations correspondent à la classe « 1 », 71 pour « 2 » et 48 pour « 3 ».

Nous utilisons la fonction to_categorical() pour créer les indicatrices :

```
#matrice y
import keras
y = keras.utils.to_categorical(D.cultivars-1)
print(y[:5,:])
```

Notons qu'il a fallu tout d'abord ramener la cible dans $\{0, 1, 2\}$ avant de procéder à l'appel de la procédure. Voici les 5 premières lignes de la matrice de dimension $(178, 3)$:

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

Une petite vérification ne peut pas faire de mal :

```
#vérification
import numpy
print(numpy.sum(y,axis=0))
```

Nous retrouvons bien la distribution des classes.

```
#print(numpy.sum(y,axis=0))
[59. 71. 48.]
```

Partition apprentissage-test. Nous partitionnons les données en 128 observations pour l'apprentissage et 50 pour le test.

```
#isoler les descripteurs
X = D.iloc[:,1:]

#subdision 50 en test (et donc 128 en apprentissage)
from sklearn import model_selection
XTrain,XTest,yTrain,yTest = model_selection.train_test_split(X,y,test_size=50,random_state=100)
```

Standardisation (centrage-réduction) des descripteurs. Les variables prédictives étant définies sur des échelles différentes, il est recommandé de les standardiser avant de procéder à un apprentissage par réseau de neurones. Nous utilisons la classe StandardScaler de la librairie scikit-learn.

```
#outil standardisation
from sklearn.preprocessing import StandardScaler

#centrage-réduction des variables
cr = StandardScaler(with_mean=True,with_std=True)
```

Nous demandons un centrage (with_mean=True) et réduction (with_std=True) des variables, c.-à-d. nous appliquons la formule :

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

Où $\bar{x}_j$ et σ_j sont respectivement la moyenne et l'écart-type de la variable X_j.

```
#calcul des paramètres + centrage réduction du train set
XTrainStd = cr.fit_transform(XTrain)

#comparaison des moyennes, avant…
print(numpy.mean(XTrain,axis=0))

#... et après CR (centrage-réduction)
print(numpy.mean(XTrainStd,axis=0))
```

fit_transform() calcule tout d'abord les paramètres de la transformation (les moyennes et écarts-type pour chaque variable), puis l'applique sur ces mêmes données XTrain.

Les moyennes avant transformation sont :

```
alcohol          13.011172
malic             2.245703
ash               2.364844
alcalinity       19.442969
magnesium        99.304688
tot_phenols       2.372031
flavonoids        2.176250
nonf_phenols      0.357500
proanth           1.633281
color             4.928828
hue               0.980125
od280             2.712656
proline         765.273438
```

Elles sont nulles (aux erreurs de troncature près) après.

```
[-6.80011603e-16  4.64905892e-16 -1.25940924e-15  4.26741975e-16
```

```
  1.38777878e-17 -5.89805982e-17 -4.82253126e-16  1.31838984e-16
 -4.16333634e-17  4.57966998e-16 -7.11236625e-17  2.98372438e-16
  0.00000000e+00]
```

4.3 Modélisation – Evaluation

Modélisation. Nous instancions un perceptron simple pour notre étude. Nous l'appliquons sur nos données d'apprentissage standardisées.

```
#keras
from keras.models import Sequential
from keras.layers import Dense
#instanciation du modèle
model = Sequential()
#architecture
model.add(Dense(units=3,input_dim=13,activation="sigmoid"))
#compilation - algorithme d'apprentissage
model.compile(loss="categorical_crossentropy",optimizer="adam",metrics=["accuracy"])
#apprentissage
model.fit(XTrainStd,yTrain,epochs=150,batch_size=10)
```

input_dim = 13 parce que 13 variables prédictives, units = 3 parce que 3 neurones (une par classe) dans la couche de sortie. La fonction de perte « loss = categorical_crossentropy » est la généralisation à la distribution multinomiale de la log-vraisemblance.

Nous affichons les poids à l'issue de l'apprentissage :

```
#poids synaptiques
print(model.get_weights())
```

Nous avons…

```
[array([[ 0.625605  , -1.4449003 , -0.00824617],
       [ 0.14056818, -0.5485751 ,  0.62404203],
       [ 0.84487724, -0.84564555,  0.43422145],
       [-0.22156928,  0.8705117 ,  0.04653621],
       [ 0.56018865, -0.42105287, -0.02743064],
       [ 0.67341274,  0.15105425, -0.44733304],
       [ 0.17756191,  0.49981773, -0.21034321],
       [-0.6545906 ,  0.20898524, -0.12943994],
       [ 0.28324762,  0.15685035, -0.06623794],
       [ 0.68136525, -1.1478251 ,  1.0142568 ],
       [ 0.7295266 ,  0.8817851 , -0.6588781 ],
       [ 0.7156769 , -0.06721698, -0.98883486],
       [  1.4496145  ,  -1.5374287  ,  -0.35039604]],  dtype=float32),  array([-
1.0351434, -1.0276651, -1.569456 ], dtype=float32)]
```

…une matrice avec 13 lignes (13 neurones de la couche d'entrée) et 3 colonnes (3 neurones de la couche de sortie) ; puis un vecteur avec 3 valeurs (lien entre le biais et les 3 neurones de la couche de sortie).

Evaluation sur l'échantillon test. Pour appliquer le modèle sur l'échantillon test, il faudrait déjà centrer et réduire les variables de ce dernier… **mais en utilisant les moyennes et écarts-type calculés sur l'échantillon d'apprentissage !** La dernière partie de la phrase est particulièrement importante. En effet, l'échantillon test préfigure la population dans laquelle sera déployée le modèle. Les observations sont considérées individuellement, indépendamment les uns des autres. Par conséquent, calculer des paramètres sur l'échantillon test n'a pas de sens.

Pour la classe StandardScaler instanciée plus haut (section 4.2), nous utilisons la méthode transform() :

```
#centrage-réduction des variables de l'échantillon test
#avec (!) les paramètres de l'éch. d'apprentissage
XTestStd = cr.transform(XTest)
```

Nous pouvons enfin faire appel à la procédure evaluate() pour mesurer les performances :

```
#évaluation
score = model.evaluate(XTestStd,yTest)
print(score)
```

Le taux de reconnaissance en test est de **96%**.

5 Conclusion

Finalement Tensorflow / Keras sont des packages comme les autres. Une fois assimilé la syntaxe des principales commandes, nous marchons sur des sentiers battus et rebattus. Le mode opératoire n'a rien de sorcier. Maintenant que nous savons les utiliser, le véritable enjeu sera de disséquer les différents algorithmes de machine learning proposés par Tensorflow : quels sont leurs caractéristiques, quels types de problèmes résolvent-ils, dans quelles situations telle ou telle approche s'applique le mieux, comment les paramétrer efficacement, etc. Hum ! ce n'est pas le boulot qui va nous manquer pour les prochaines semaines…

6 Références

- Keras : The Python Deep Learning Library (https://keras.io/).
- Tensorflow : An Open-Source Machine Learning Framework for Everyone (https://www.tensorflow.org/).
- (RAK, 2013) Ricco Rakotomalala, « Paramétrer le perceptron multicouche », avril 2013.
- Karlijn Willems, « Keras Tutorial : Deep Learning in Python », May 2017 ; https://www.datacamp.com/community/tutorials/deep-learning-python
- Jason Brownlee, « Develop Your First Neural Network in Python With Kears Step-By-Step », May 2016.