TP RESEAUX DE NEURONES ARTIFICIELS EN PYTHON

Master I : SID

Auteur: Khadir Mohamed Tarek

Préambule:

Ce document introduit l'implémentation des réseaux de neurones artificiels, en particulier le Perceptron Multi Couche (PMC) sous python en utilisant la librairie Scikit-Learn. La théorie sur les RNA et l'apprentissage par rétro propagation du gradient, ne sera pas relaté dans ce document technique, et l'étudiant devra se référer pour la théorie au support de cours.

1. Scikit-Learn

Pour suivre ce tutoriel, vous devez installer la dernière version de SciKit Learn (> 0.18)! Il est facile à installer via pip ou conda, mais vous pouvez vous référer à la documentation d'installation officielle pour obtenir des détails complets à ce sujet.

2. Conda et IPython Notebook (Jupyter Notebook)

Conda est un système de gestion de packges open source et un système de gestion de l'environnement fonctionnant sous Windows, macOS et Linux. Conda installe, exécute et met à jour rapidement les packages et leurs dépendances. Conda crée, enregistre, charge et commute facilement entre les environnements de votre ordinateur local. Il a été créé pour les programmes Python, mais il peut empaqueter et distribuer des logiciels pour n'importe quelle language.

Conda en tant que gestionnaire de paquets vous aide à trouver et à installer des packges. Si vous avez besoin d'un package nécessitant une version différente de Python, vous n'avez pas besoin de passer à un autre gestionnaire d'environnement, car conda est également un gestionnaire d'environnement. Avec seulement quelques commandes, vous pouvez configurer un environnement totalement séparé pour exécuter cette version différente de Python, tout en continuant d'exécuter votre version habituelle de Python dans votre environnement normal.

Dans sa configuration par défaut, conda peut installer et gérer les milliers de paquets créés sur repo.continuum.io qui sont générés, examinés et gérés par Anaconda®.

Etapes d'installation de Conda :

Windows

1. Désactivez tout anti-virus (tels Norton, McAfee, Avast, etc.) pour le temps de l'installation;
2. Allez à l'adresse http://continuum.io/downloads;
3. Choisir votre OS en cliquant sur l'icone correspondant (si ce n'est déjà fait);
4. Cliquez sur le lien « I WANT PYTHON 3.4 »;
5. Téléchargez l'installateur en cliquant sur le lien « Windows 64-Bit Python 3.4 Graphical Installer »;
6. Double cliquez sur le « .exe » de l'installateur pour procéder à l'installation;
7. Ouvrez une fenêtre de commande (command prompt);
8. Testez votre installation en exécutant la commande « ipython ».

Un moyen facile d'obtenir SciKit-Learn et tous les outils dont vous avez besoin pour faire cet exercice consiste à utiliser le logiciel iPython Notebook de Anaconda. Ce tutoriel vous aidera à utiliser ces outils pour vous permettre de créer un réseau de neurones en Python.

IPython a pour objectif de créer un environnement complet pour l'informatique interactive et exploratoire. Pour atteindre cet objectif, IPython comprend trois composants principaux:

- Un shell Python interactif amélioré.
- Un modèle de communication découplé à deux processus, qui permet à plusieurs clients de se connecter à un noyau de calcul, notamment le bloc-notes Web fourni avec Jupyter.
- Une architecture pour le calcul parallèle interactif qui fait maintenant partie du paquage ipyparallel.

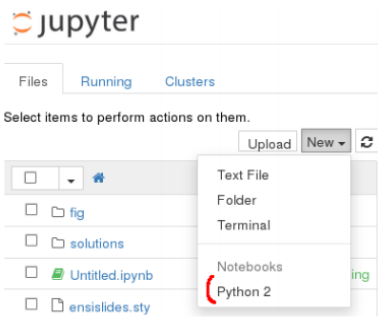
Intérêts :

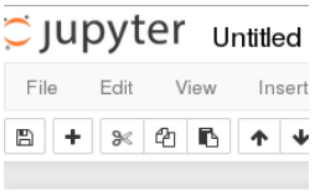
- Environnement interactif et agréable
- Pratique pour dérouler un calcul/raisonnement en mélangeant code et explications I Possibilité d'utiliser un serveur "notebook" et de s'y connecter de n'importe quelle machine (sans Python installé)
- Le serveur "notebook" peut être le point d'entrée d'une ferme de calcul.

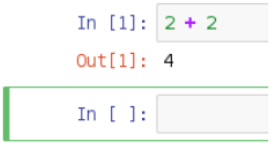
Premiers Pas :

- Linux : ipython notebook ou jupyter notebook
- WinPython : lancer Jupyter ou IPython Notebook.
- Anaconda : depuis le launcher anaconda, choisir Jupyter ou IPython Notebook.

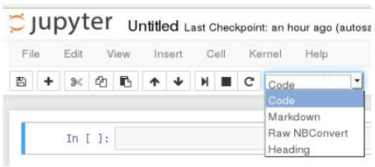
Utilisation : depuis le navigateur (doit se lancer automatiquement)
Nouveau notebook : New → Notebook/Python 2. Entrer 2+2, puis Shift+Enter







Organiser un Notebook :



Code Python
Markdown Texte formaté :

- `# Titre 1`
- `## Titre 2`
- `**gras**, *italique*, 'code', $formule LaTeX$`

```
Essayez : Ceci texte est en *italique* et celui-ci en **gras**. $\sin(\pi)$
en Python = 'math.sin(math.pi)'.
```

Sauvegarder et restaurer son travail

- Par défaut : autosave (automatique) + checkpoint (clic sur le bouton)
- Enregistré à l'endroit où Notebook a été lancé (éventuellement sur serveur distant)
- Autre option :

    1. menu « File » → « Download as » pour récupérer le notebook sous forme de fichier ipynb.
    2. menu « File » → « Open », puis « Upload » pour envoyer un fichier ipynb au notebook.

- Essayez :
    1. Importez notebook-et-markdown.ipynb dans votre notebook.
    2. Modifiez-le, puis téléchargez-le dans différent formats.

3. Données Utilisées

Les données utilisées pour cet exemple concernent une BDD regroupant les caractéristiques d'un produit pharmaceutique ainsi que des producteurs qui le fabriquent. Le rôle du RNA qu'on va concevoir est de définir le producteur à partir des caractéristiques des produits (leurs caractéristiques chimiques) présentées. Les données peuvent être trouvées dans le fichier produc_data.csv.

4. Procédure

Tout d'abords il faudra installer conda, puis appeler jupyter notebook (iPython Notebook). Un guide pour commencer avec le notebook et l'écriture de script python est donné par document notebook.

Commençons par importer le jeu de données! Nous allons utiliser la fonctionnalité de noms de Pandas pour nous assurer que les noms de colonne associés aux données sont bien importés.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import datetime
        %matplotlib inline
        plt.rcParams['figure.figsize'] = [12,8]
```

Afin d'importer les donnes à partir du fichier produc_data.csv, la fonction pd.read_csv est invoqué comme suit :

product = pd.read_csv('product_data.csv', names = ["Producer", "Alchol", "Malic_Acid", "Ash", "Alcalinity_of_Ash", "Magnesium", "Total_phenols", "Falvanoids", "Nonflavanoid_phenols", "Proanthocyanins", "Color_intensity", "Hue", "OD280", "Proline"])

Définissant la désignation de chaque composant de chaque colonne.

```
In [2]: import pandas as pd
        product = pd.read_csv('product_data.csv', names = ["Producer", "Alchol", "Malic_Acid", "Ash", "Alcalinity_of_Ash", "Magnesium", "
```

Afin de visualiser et de s'assurer que la BDD à bien été chargé, la fonction . head est utilisée.

```
In [3]: product.head()
```

Out[3]:

	Producer	Alchol	Malic_Acid	Ash	Alcalinity_of_Ash	Magnesium	Total_phenols	Falvanoids	Nonflavanoid_phenols	Proanthocyanins	Color_intensity	Hue
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04

La BDD chargée dans la variable « product » peut être aussi visualisée autrement en utilisant la fonction .describe() et la fonction .transpose(). Cette fonction nous donne les caractéristiques des donées contenu dans « product » : moyenne, standart deviation, min, max, etc.

```
In [4]: product.describe().transpose()
```

Out[4]:

	count	mean	std	min	25%	50%	75%	max
Producer	178.0	1.938202	0.775035	1.00	1.0000	2.000	3.0000	3.00
Alchol	178.0	13.000618	0.811827	11.03	12.3625	13.050	13.6775	14.83
Malic_Acid	178.0	2.336348	1.117146	0.74	1.6025	1.865	3.0825	5.80
Ash	178.0	2.366517	0.274344	1.36	2.2100	2.360	2.5575	3.23
Alcalinity_of_Ash	178.0	19.494944	3.339564	10.60	17.2000	19.500	21.5000	30.00
Magnesium	178.0	99.741573	14.282484	70.00	88.0000	98.000	107.0000	162.00
Total_phenols	178.0	2.295112	0.625851	0.98	1.7425	2.355	2.8000	3.88
Falvanoids	178.0	2.029270	0.998859	0.34	1.2050	2.135	2.8750	5.08
Nonflavanoid_phenols	178.0	0.361854	0.124453	0.13	0.2700	0.340	0.4375	0.66
Proanthocyanins	178.0	1.590899	0.572359	0.41	1.2500	1.555	1.9500	3.58
Color_intensity	178.0	5.058090	2.318286	1.28	3.2200	4.690	6.2000	13.00
Hue	178.0	0.957449	0.228572	0.48	0.7825	0.965	1.1200	1.71
OD280	178.0	2.611685	0.709990	1.27	1.9375	2.780	3.1700	4.00
Proline	178.0	746.893258	314.907474	278.00	500.5000	673.500	985.0000	1680.00

La fonction .shape() permet d'afficher un tuple qui donne la dimensionnalité des données.

```
In [5]: # 178 data points with 13 features and 1 label column
        product.shape
```

```
Out[5]: (178, 14)
```

Afin de définir les entrées et sorties du réseau de neurones Artificiel les fonctions .drop() et une simple affectation du vecteur contenant les sorties sont utilisées.

```
In [6]: X = product.drop('Producer',axis=1)
        y = product['Producer']
```

Une fois que les données d'entrées et de sorties défini on peut commencer la conception et l'apprentissage du réseau de neurones Artificiels. La fonction sklearn.model_selection . La fonction et sa syntaxe permet d'obtenir aléatoirement à partir de matrices, des bases d'apprentissage et de validation.

```
In [7]: from sklearn.model_selection import train_test_split
```

```
In [8]: X_train, X_test, y_train, y_test = train_test_split(X, y)
        #X_train.shape
        #X_test.shape
        #y_train.shape
        y_test.shape
```

```
Out[8]: (45,)
```

La normalisation des jeux de données est une exigence commune à de nombreux estimateurs d'apprentissage automatique implémentés dans scikit-learn; ils pourraient se comporter mal si les caractéristiques individuelles ne ressemblent pas plus ou moins aux données standards normalement distribuées: gaussienne avec moyenne nulle et variance d'unité. Une fois obtenu, les données d'apprentissages et de validation devront être épurées et normalisées. Ceci se fait en utilisant la fonction Sklearn.preprocessing.

```
In [10]: from sklearn.preprocessing import StandardScaler

In [11]: scaler = StandardScaler()
```

La fonction .fit(), permet de compléter la standardisation des données.

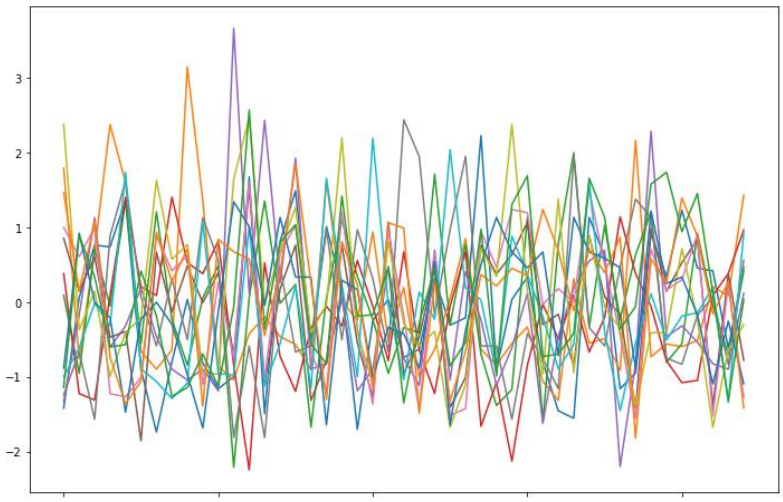
```
In [12]: # Fit only to the training data
         scaler.fit(X_train)
Out[12]: StandardScaler(copy=True, with_mean=True, with_std=True)

In [13]: StandardScaler(copy=True, with_mean=True, with_std=True)
Out[13]: StandardScaler(copy=True, with_mean=True, with_std=True)
```

Maintenant on peut appliquer les transformations pour les données d'apprentissage en utilisant la fonction .transform(). Qui permet d'uniformiser la taille des données.

```
In [14]: # Now apply the transformations to the data:
         X_train = scaler.transform(X_train)
         X_test = scaler.transform(X_test)
```

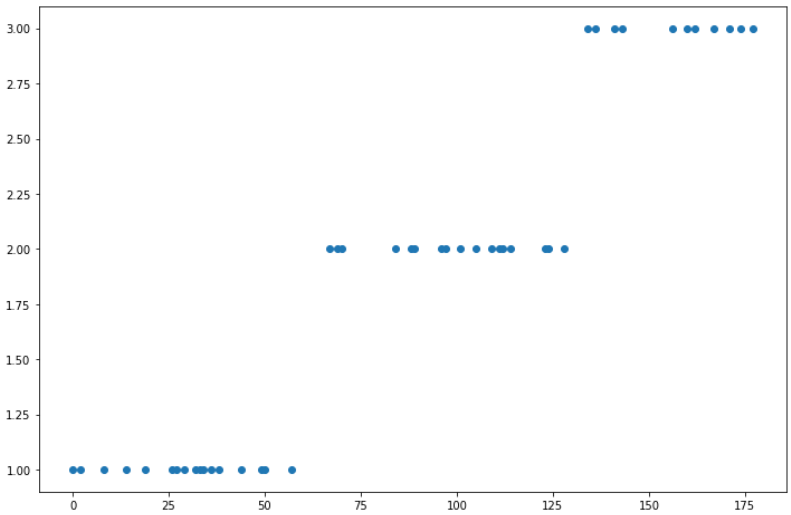
On a maintenant spécifié les données d'apprentissage et de test (entrées et sorties). On peut utiliser la fonction plt.plot() pour afficher les données sélectionnés, dans le cas suivant les données entrées pour le test (13 variables sur 45 échantillons) sélectionnées aléatoirement.

```
In [15]: plt.plot(X_test, label='TEST SET')
         X_test.shape
Out[15]: (45, 13)
```



De la même manière on peut afficher les données de sorties utilisés pour le test (y_test). A noter que l'affichage ici est fait par rapport au coordonnées du jeux de données initiale.

```
In [16]: plt.plot(y_test, 'o', label='TEST SET')
         y_test.shape
Out[16]: (45,)
```



On peut alors construire un réseaux de neurones de type MLP en utilisant la fonction MLPClassifier() . Ici on construit un MLP de 13-13-13, c.a.d 13 entrées, 13 neurones dans la première couche cachée et 13 neurones dans la deuxième couche cachée, avec un maximum d'itération (époques) égale à 700.

```
In [18]: mlp = MLPClassifier(hidden_layer_sizes=(13,13,13),max_iter=700)
```

L'apprentissage du réseau spécifié en utilisant les données préalablement choisies, se fait en utilisant la fonction mlp.fit() qui prend comme argument, au mois, les données d'apprentissage d'entrées et de sortie. L'apprentissage terminé, la fonction retourne les résultats comme suit.

```
In [19]: mlp.fit(X_train,y_train)
Out[19]: MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
                beta_2=0.999, early_stopping=False, epsilon=1e-08,
                hidden_layer_sizes=(13, 13, 13), learning_rate='constant',
                learning_rate_init=0.001, max_iter=700, momentum=0.9,
                nesterovs_momentum=True, power_t=0.5, random_state=None,
                shuffle=True, solver='adam', tol=0.0001, validation_fraction=0.1,
                verbose=False, warm_start=False)
```

Afin de valider notre apprentissage il faudra évaluer les résultats sur les données de test. La fonction mlp.predict(). La fonction prend en argument les données d'entrées de test, et retourne les sorties du RNA

```
In [20]: predictions = mlp.predict(X_test)
In [21]: predictions
Out[21]: array([2, 1, 1, 3, 3, 2, 2, 2, 2, 3, 2, 2, 1, 2, 1, 1, 2, 3, 1, 2, 3, 1, 2,
                2, 1, 3, 3, 1, 2, 1, 1, 2, 2, 2, 1, 1, 2, 3, 1, 1, 1, 1, 3, 2, 3], dtype=int64)
In [22]: predictions[:10]
Out[22]: array([2, 1, 1, 3, 3, 2, 2, 2, 2, 3], dtype=int64)
```

Afin de représenter les sorties réels et les sorties prédites les données doivent êtres organiser de la même manière. Les données « y_test » sont organisés en abscices et ordonnées, alors que les résultats « prédictions » sont juste un vecteur de résultats avec les abscices de 1 à 45. La fonction .head() donne la structure des données y_test.

```
In [23]: y_test.head()
Out[23]: 124    2
         38     1
         19     1
         143    3
         177    3
         Name: Producer, dtype: int64
```

Afin d'uniformiser les résultats « predictions » en leur rajoutant les abscices de « y_test », la fonction pd.Series(). La fonction prend en argument les données predictions et lui rajoute l'index de y_test.

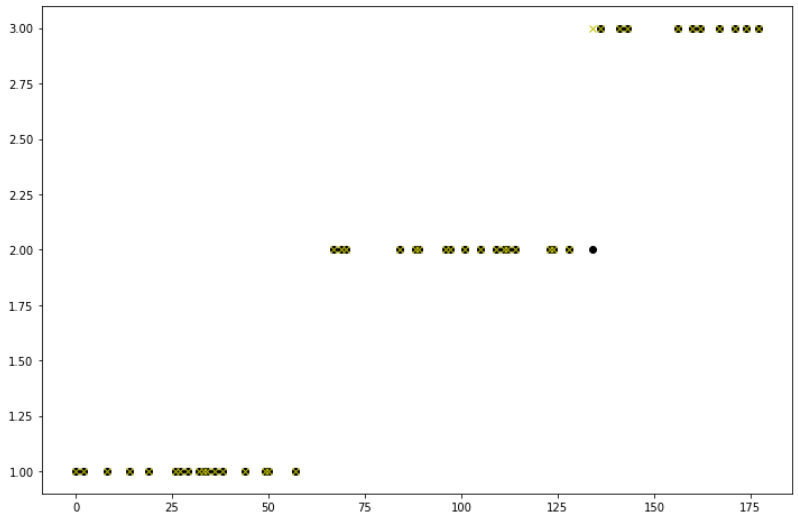
```
In [24]: spredictions = pd.Series(data=predictions, index=y_test.index)
```

Le résultat obtenu « sprediction » a maintenant la meme indexation que y_test.

```
In [25]: spredictions.index
Out[25]: Int64Index([124,  38,  19, 143, 177,  70, 109, 128, 123, 160, 114,  69,  14,
                      96,  32,   0, 101, 156,   2,  88, 167,  44, 105, 134,  36, 171,
                     162,   8,  97,  50,  57,  67,  84, 112,  49,  34,  89, 136,  33,
                      27,  29,  26, 141, 111, 174],
                    dtype='int64')

In [26]: y_test.index
Out[26]: Int64Index([124,  38,  19, 143, 177,  70, 109, 128, 123, 160, 114,  69,  14,
                      96,  32,   0, 101, 156,   2,  88, 167,  44, 105, 134,  36, 171,
                     162,   8,  97,  50,  57,  67,  84, 112,  49,  34,  89, 136,  33,
                      27,  29,  26, 141, 111, 174],
                    dtype='int64')
```

Nous pouvons maintenant afficher et comparer graphiquement les résultats et les données réelles. La fonction plt.plot(), déjà abordée, est utilisé.

```
In [27]: plt.plot(spredictions,'ko',y_test,'yx')
         predictions.shape
Out[27]: (45,)
```



Afin d'évaluer plus en détails les résultats obtenus, la matrice de confusion « confusion matrix » peut être calculé a cet effet. La matrice de confusion calcule les vrais positifs, les faux positifs, les vrais négatifs et les faux négatifs. La matrice de confusion est importée à partir de sklearn-metrics comme suit :

```
In [28]: from sklearn.metrics import classification_report,confusion_matrix
```

La matrice de confusion est donnée dans ce qui suit :

```
In [30]: print(confusion_matrix(y_test,spredictions))
```

```
[[17  0  0]
 [ 0 17  0]
 [ 0  1 10]]
```

Un rapport sur la classification avec le calcul de la précisions recall et fallout basé sur la matrice de confusion est donnée par la [fonction classification_report()](#).

```
In [31]: print(classification_report(y_test,spredictions))
```

	precision	recall	f1-score	support
1	1.00	1.00	1.00	17
2	0.94	1.00	0.97	17
3	1.00	0.91	0.95	11
avg / total	0.98	0.98	0.98	45