1 Introduction

Mise en œuvre des algorithmes de descente de gradient stochastique avec Python. Utilisation du package « scikit-learn ».

Ce tutoriel fait suite au support de cours consacré à l'application de la méthode du gradient en apprentissage supervisé (RAK, 2018). Nous travaillons sous Python. Un document similaire a été écrit pour le logiciel R dans le cadre de la régression linéaire multiple.

Nous travaillons sur un problème de classement cette-fois. Nous souhaitons estimer les paramètres de la régression logistique à partir d'un ensemble de données étiquetées. Nous utilisons le package « scikit-learn » particulièrement populaire auprès des aficionados de Python[1]. Nous étudierons l'influence du paramétrage sur la rapidité de la convergence de l'algorithme d'apprentissage et, de manière plus générale, sur la qualité du modèle obtenu. Nous en profiterons pour détailler une petite curiosité, parce que peu mise en avant dans les supports, que constitue la construction de la courbe ROC (Receiver Operating Characteristic) en validation croisée.

2 Données

Nous utilisons le *dataset* « sonar » bien connu des data scientists. Il est accessible sur le dépôt de l'UCI Machine Learning Repository[2]. L'objectif est de reconnaître le type d'un objet (y = 1 = mine vs. y = 0 = rock) à partir de signaux recueillis à l'aider d'un sonar. Les variables prédictives (x1, …, x60) sont normalisées. Elles varient entre 0 et 1. Puisqu'elles sont sur la même échelle, nous sommes dispensés de la phase de préparation de données.

3 Descente de gradient stochastique

3.1 Importation et inspection des données

Nous importons les données avec la procédure **read_table()** de la librairie **Pandas**.

[1] KDnuggets Tutorials – Overviews, « Top 20 Python AI and Machine Learning Open Source Projets », February 2018.
[2] http://archive.ics.uci.edu/ml/datasets/connectionist+bench+(sonar,+mines+vs.+rocks)

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

#importation des données
import pandas
D = pandas.read_table("sonar.txt",delimiter="\t",header=0)
```

Nous vérifions les dimensions et la liste des variables.

```
#dimensions
print(D.shape)

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

Nous avons 208 observations et 61 variables, dont la variable cible y.

```
(208, 61)
Index(['x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x10', 'x11',
       'x12', 'x13', 'x14', 'x15', 'x16', 'x17', 'x18', 'x19', 'x20', 'x21',
       'x22', 'x23', 'x24', 'x25', 'x26', 'x27', 'x28', 'x29', 'x30', 'x31',
       'x32', 'x33', 'x34', 'x35', 'x36', 'x37', 'x38', 'x39', 'x40', 'x41',
       'x42', 'x43', 'x44', 'x45', 'x46', 'x47', 'x48', 'x49', 'x50', 'x51',
       'x52', 'x53', 'x54', 'x55', 'x56', 'x57', 'x58', 'x59', 'x60', 'y'],
      dtype='object')
```

Nous séparons les descripteurs de la variable cible dans deux structures distinctes.

```
#convertir en structure matrice numpy
D = D.as_matrix()

#variables prédictives jusqu'à l'avant-dernière : matrice
X = D[:,:-1]
print(X.shape) #(208, 60)

#variable cible - la dernière : vecteur
y = D[:,-1]
print(y.shape) #(208,)
```

La variable cible y est binaire {1, 0}. Nous comptabilisons le nombre d'observations dans chaque classe.

```
#numpy
import numpy
```

```
#décompte des effectifs des classes
print(numpy.unique(y,return_counts=True))
```

Nous avons 97 observations de la classe 0, 111 de la classe 1.

```
(array([0., 1.]), array([ 97, 111], dtype=int64))
```

3.2 Descente de gradient stochastique

3.2.1 Librairie scikit-learn

Nous souhaitons utiliser la librairie « **scikit-learn** ». La première chose à faire est de vérifier la version disponible. En effet, les paramètres peuvent différer d'une version à l'autre. Certains deviennent obsolètes. D'autres apparaissent, avec des valeurs par défaut qui influent sur le comportement de l'algorithme.

```
#version de scikit-learn !!! important pour les options
import sklearn
print(sklearn.__version__)
```

```
0.19.1
```

Nous disposons de la version **0.19.1** pour ce tutoriel (01/05/2018).

3.2.2 Modélisation.

Nous importons la classe SGDClassifier que nous instancions comme suit :

```
#initialisation
from sklearn.linear_model import SGDClassifier
msgd1 = SGDClassifier(loss="log",penalty="none",learning_rate="constant",
                      eta0=0.1,max_iter=100,tol=None)
```

Précisons les paramètres :

- La fonction de perte (loss = "log") correspond à celle de la régression logistique. Nous y reviendrons plus en détail plus bas.
- Nous utilisons une régression non-régularisée (penalty="none"). Les autres alternatives sont les régressions ridge, lasso ou elasticnet.
- Le taux d'apprentissage η sera constant tout au long du processus (learning_rate = "constant").
- Il sera égal à $\eta = \eta_0 = 0.1$ (eta0 = 0.1).

– Le nombre maximum d'itérations, nombre de passage sur la totalité de la base, est égal à 100 (max_iter = 100).
– Nous n'utilisons pas la règle d'arrêt basée sur l'évolution de la fonction de perte (tol = None).

Nous lançons le processus d'apprentissage avec la fonction **fit()**

```
#modélisation
msgd1.fit(X,y)
```

Python précise les paramètres utilisés. Nous pouvons visualiser les valeurs par défaut des paramètres que nous n'avons pas explicitement spécifiés.

```
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.1, fit_intercept=True, l1_ratio=0.15,
       learning_rate='constant', loss='log', max_iter=100, n_iter=None,
       n_jobs=1, penalty='none', power_t=0.5, random_state=None,
       shuffle=True, tol=None, verbose=0, warm_start=False)
```

A l'issue du processus, nous pouvons afficher le nombre d'itérations réellement effectués.

```
#nombre d'itération
print(msgd1.n_iter_) # 100
```

Avec notre paramétrage, le nombre d'itérations correspond à la valeur de max_iter, soit 100.

Que vaut notre modèle ?

3.2.3 Fonction de perte

La fonction de perte log loss correspond, à un facteur multiplicatif près, à la log-vraisemblance utilisée pour la régression logistique (RAK, 2011). Elle s'écrit :

$$J = -\frac{1}{n}\sum_{i=1}^{n} y_i \log p_i + (1 - y_i)\log(1 - p_i)$$

Où : n est le nombre d'observations, y_i est la valeur de la cible pour l'individu n°i, p_i est la probabilité conditionnelle estimée $P(y_i=1/X)$ fournie par le modèle.

Calcul avec le module metrics. Le module **metrics** de scikit-learn peut la fournir. La fonction **log_loss**() prend en entrée les étiquettes observées y_i et les probabilités estimées p_i.

Calculons tout d'abord ces dernières sur notre échantillon de données. Nous utilisons la fonction **predict_proba()**.

```
#probabilités conditionnelles
probas = msgd1.predict_proba(X)
print(probas[:10,:])
```

A l'affichage des 10 premières valeurs, nous constatons qu'il y a 2 colonnes de probabilités complémentaires. La première correspond à P(y=0/X), la seconde à P(y=1/X). C'est cette dernière qui nous intéresse.

```
[[0.39390876 0.60609124]
 [0.85451437 0.14548563]
 [0.08418015 0.91581985]
 [0.45856191 0.54143809]
 [0.53152366 0.46847634]
 [0.9612033  0.0387967 ]
 [0.97567256 0.02432744]
 [0.04818839 0.95181161]
 [0.27892305 0.72107695]
 [0.55074028 0.44925972]]
```

Nous faisons maintenant appel à la fonction **log_loss()** du module **metrics**.

```
#valeur de la fonction de perte
from sklearn import metrics
logloss = metrics.log_loss(y,probas[:,1],normalize=True,eps=1e-15)
print(logloss)
```

Nous obtenons :

```
0.33265103465822143 (J1)
```

La valeur en elle-même n'est pas très intéressante. Elle nous servira surtout de repère pour situer les variantes de SGDClassifier que nous implémenterons avec d'autres paramétrages.

Calcul explicite. Lors de l'appel de **log_loss()**, deux paramètres (normalize) et (eps) dont les valeurs par défaut sont respectivement (True) et (1e-15) attirent notre attention. Le premier signifie que nous calculons bien la moyenne de la perte c.-à-d. nous appliquons le coefficient $\frac{1}{n}$ dans le calcul de J. Le second indique que les probabilités trop proches de 1 ou de 0 sont corrigées avec le facteur eps.

Vérifions que nous obtenons bien le résultat ci-dessus en explicitant les étapes du calcul. Nous corrigeons tout d'abord les probabilités conditionnelles extrêmes.

```
#seuils pour probas
eps = 1e-15

#probabilités conditionnelles P(Y=1/X)
p = probas[:,1]

#corriger les probas
p[p<eps] = eps
p[p>(1-eps)] = 1-eps
```

Puis nous appliquons la formule du log_loss

```
#calcul manuel
import numpy
print(numpy.mean(-(y*numpy.log(p)+(1-y)*numpy.log(1-p))))
```

Nous obtenons exactement la même valeur J_1 = 0.33265103465822143.

3.2.4 Critère AUC en validation croisée

Nous souhaitons utiliser l'AUC (Area Under Curve – Aire sous la courbe) de la courbe ROC (Receiver Operating Characteristic) pour évaluer notre modèle. Nous savons qu'apprécier le modèle sur les données ayant servi à le construire amène une vision biaisée de ses performances en favorisant les modèles les plus complexes. La faible taille de notre base (208 observations) ne nous permet cependant pas de la scinder en échantillons d'apprentissage et de test pour un schéma de type « holdout » classique (apprentissage – test). Nous nous tournons donc vers la validation croisée qui permet, rappelons-le, d'estimer les performances du modèle construit sur la totalité des données disponibles.

La description du calcul de l'AUC en validation croisée n'est pas très répandue dans la littérature. Il en est également de même en ce qui concerne la courbe ROC. Nous consacrerons une section spécifique à cette problématique plus bas (section 0). Pour l'heure, nous nous attachons simplement à évaluer les différents modèles en laissant scikit-learn calculer l'AUC pour nous.

Tout d'abord, nous initialisons un objet du type **StratifiedKFold** qui permet de réaliser une validation croisée avec un échantillonnage stratifié c.-à-d. les proportions des classes sont respectées dans les blocs (*folds*).

```
#AUC en validation croisée
from sklearn import model_selection

#préparer la constitution en groupes - pour avoir les mêmes à chaque lancement
cvEval = model_selection.StratifiedKFold(n_splits=3,shuffle=True,random_state=1)
```

Nous demandons une subdivision en K = 3 blocs (n_splits = 3), les observations sont mélangées aléatoirement avant d'être subdivisées (shuffle = True), nous fixons la valeur de départ du générateur de nombres aléatoires pour que la démarche soit reproductible au fil de l'expérimentation (random_state = 1).

Remarque : Nous créons un objet spécifique de validation croisée dans notre expérimentation pour que nous puissions réutiliser les mêmes *folds* dans les évaluations des différents modèles que nous instancierons. Les résultats seront directement comparables.

Nous lançons la procédure de validation croisée en lui passant en paramètre l'objet cvEval.

```
#AUC en validation croisée
auc = model_selection.cross_val_score(msgd1,X,y,cv=cvEval,scoring='roc_auc')
```

Outre cvEval, les paramètres sont :

- L'objet de modélisation prédictive (msgd1) avec ses paramètres.
- La matrice des descripteurs (X) et le vecteur des étiquettes (y).
- La mesure d'évaluation (scoring = ''auc_roc").

La variable auc est un vecteur qui contient 3 valeurs.

```
#valeurs AUC
print(auc)
[0.86076986 0.78969595 0.7660473 ]
```

Ce qui est tout à fait logique puisque nous avons demandé une validation croisée en 3 *folds*. Le processus « apprentissage – test » a été répété 3 fois. Nous avons une valeur de l'AUC pour chaque exécution (RAK, 2015).

La question maintenant est de produire une mesure synthétique. Nous choisissons d'utiliser une moyenne non pondérée. Nous pouvons lire l'AUC comme suit : dans la population, un individu quelconque appartenant à la classe (y = 1) a 80.55% de chances de se voir attribuer une probabilité conditionnelle P(y= 1 / X) supérieure à celle d'un individu (y = 0).

```
#AUC moyenne
print(auc.mean())
```

```
0.805504368004368 = AUC1
```

Remarque : En réalité, le choix de la moyenne non pondérée peut être discuté. Les *folds* n'ont pas exactement les mêmes effectifs puisque 208 n'est pas divisible par 3 (après vérifications, nous avons [70, 69, 69]). Une moyenne tenant compte du poids des *folds* est une option alternative. De même, nous verrons cela plus bas, il est possible de construire tout d'abord une courbe ROC « moyenne » puis de calculer son AUC. Quoiqu'il en soit, les différences entre les valeurs synthétiques obtenues restent marginales.

3.3 Ajustement des paramètres

Les méthodes basées sur la descente de gradient sont très sensibles aux paramètres, notamment ceux qui pèsent sur la vitesse de convergence et la qualité de l'optimisation. Dans cette section, nous procédons à quelques modifications et nous observons leurs conséquences sur le comportement du modèle.

3.3.1 Fonction pour l'évaluation des modèles

Les comparaisons étant assez répétitives, nous avons choisi de grouper dans une fonction le calcul de la perte "log_loss" et de l'AUC en validation croisée. Elle prend en paramètres : l'objet modèle à évaluer (classifier), l'objet validation croisée permettant de reproduire à l'identique les blocs « apprentissage – test » (cv), la matrice des descripteurs (X), le vecteur des étiquettes (y).

```
#fonction pour évaluer un classifieur
def evaluation(classifier,cv,X,y):
    #learning process
    classifier.fit(X,y)
    #nombre d'itérations
    print("ITERATIONS = ",classifier.n_iter_)
```

```
    #get the probabilities
    p = classifier.predict_proba(X)[:,1]
    #loss score
    ls = metrics.log_loss(y,p,normalize=True,eps=1e-15)
    print("LOSS = ",ls)
    #auc
    auc = model_selection.cross_val_score(classifier,X,y,cv=cv,scoring='roc_auc')
    print("AUC = ", auc.mean())
    #
    return classifier.n_iter_, ls, auc.mean()
```

Elle affiche et renvoie le nombre d'itérations, la valeur de la fonction de perte et l'AUC en validation croisée.

3.3.2 Règles d'arrêt basée sur l'évolution de la fonction de perte

La première tentative consiste à modifier la règle d'arrêt. Plutôt que de spécifier le nombre d'itérations (max_iter = None), nous décidons le processus lorsque la perte ne décroît plus de manière significative, soit

$$J^t > J^{t-1} + tol$$

Le numéro d'itération t correspond ici au passage de l'ensemble des observations de la base.

Nous fixons (tol = 1e-5) :

```
#modifier la règle d'arrêt
msgd2 = SGDClassifier(loss="log",penalty="none",learning_rate="constant",
                      eta0=0.1, max_iter=None,tol=1e-5)

#evaluation
evaluation(msgd2,cvEval,X,y)
```

Nous obtenons :

```
ITERATIONS =  8
LOSS =  0.4354899452040547
AUC =  0.8001893939393939
```

Le nombre d'itérations est nettement plus faible (itérations = 8). Le temps de calcul est fortement réduit. En contrepartie, l'optimisation est moins aboutie avec J2 = 0.4355 (vs. J1 = 0.3326, section 3.2.3). Mais cela a une d'incidence négligeable sur la capacité à discriminer (AUC2 = 80.02%).

3.3.3 Décroissance du taux d'apprentissage

Une autre piste d'amélioration est de faire évoluer le taux d'apprentissage durant le processus. Nous commençons avec une valeur élevée pour accéder rapidement à la zone d'optimalité, nous finissons avec une valeur faible pour être plus précis dans la détermination de la solution définitive. La formule de décroissante s'écrit :

$$\eta_t = \frac{\eta_0}{t^{0.25}}$$

Le coefficient 0.25 est lui-même éventuellement modifiable. Nous ne l'avons pas fait ici.

Voyons ce qu'il en est avec $\eta_0 = 0.1$:

```
#décroissance de eta
msgd3 = SGDClassifier(loss="log",penalty="none",learning_rate="invscaling",
                      eta0=0.1, max_iter=None,tol=1e-5)

#evaluation
evaluation(msgd3,cvEval,X,y)
```

Le nombre d'itérations est naturellement plus élevé, sans pour autant améliorer la précision et les performances. C'est même décevant en réalité.

```
ITERATIONS =  86
LOSS =  0.551554955726462
AUC =  0.7946355446355446
```

3.3.4 Paramétrage par défaut

Après plusieurs tentatives infructueuses, je me suis demandé ce que pouvait bien donner le paramétrage par défaut de la méthode. Après tout, j'imagine que les concepteurs de l'outils les ont déterminés pour être efficaces sur une plage d'application relativement large. Nous conservons bien sûr la fonction de perte et l'absence de régularisation.

```
#paramétrage par défaut
msgd4 = SGDClassifier(loss="log",penalty="none")

#evaluation
evaluation(msgd4,cvEval,X,y)
```

Le nombre d'itérations est limité à 5 comme indiqué dans la documentation (scikit-learn, version 0.19). Nous sommes assez loin du compte en termes de log-loss (J4 = 2.1375…). Les capacités de discrimination sont légèrement dégradées avec AUC = 77.86%.

```
ITERATIONS =  5
LOSS =  2.1375776971893714
AUC =  0.7786309036309037
```

Remarque : Tout ça paraît logique. Il reste quand même que les résultats sont assez instables, même en validation croisée, sur une base de taille aussi réduite (208 observations). En effet, le processus de descente de gradient stochastique garde une part aléatoire notamment lors du mélange (*shuffle*) de l'ordre des individus (il faudrait manipuler le paramètre random_state de SGDClassifier pour le contrôler également). Nous avons parfois obtenu des valeurs légèrement différentes lors de la réexécution de l'expérimentation.

Nous optons pour le second modèle "msgd2" pour la suite.

4 Courbe ROC en validation croisée

Rappelons brièvement le principe de la K-folds validation croisée (RAK, 2015). Les données sont subdivisées en K blocs d'effectifs égaux (aussi identiques que possible tout du moins si la taille d'échantillon "n" n'est pas divisible par K). La procédure suivante est répétée K fois : le modèle est construit sur (K-1) blocs (apprentissage) puis il est évalué sur le K^{ème} bloc (test) ; nous faisons tourner les blocs pour obtenir K valeurs de la mesure de performances. L'estimation en validation croisée est une mesure synthétique déduite de ces K valeurs.

Le calcul de l'AUC en validation croisée m'a assez intrigué parce qu'on la rencontre peu dans la littérature. Quand le nombre de *folds* est égal à 3, cela veut dire que 3 courbes ROC sont implicitement construites pour obtenir 3 valeurs de l'AUC. Dans cette section, en nous inspirant d'un exemple de la documentation en ligne de scikit-learn, nous allons les construire explicitement pour en tirer une courbe synthétique à partir de laquelle nous extrairons la valeur de l'AUC. Nous verrons alors comment la situer par rapport à celle fournie par la fonction **cross_val_score()** du module **metrics**.

Il s'agit d'itérer sur les paires d'identifiants d'individus « apprentissage-test » correspondant aux (K-1) et K^{ème} blocs. Nous disposerons ainsi de K courbes ROC matérialisées par les couples de valeurs fpr (*false positive rate*, taux de faux positifs) en abscisse et tpr (*true positive rate*, taux de vrais positifs) en ordonnée. Nous les collectons dans une liste **lstRoc**.

```
#liste pour construction des courbes ROC
lstRoc = []

#courbe ROC en validation croisée
for idTrain,idTest in cvEval.split(X,y):
    #modélisation sur les individus idTrain, (K-1) blocs
    msgd2.fit(X[idTrain,:],y[idTrain])
    #probabilités d'appartenir à la classe 1 sur idTest (Kème bloc)
    p = msgd2.predict_proba(X[idTest,:])[:,1]
    #valeurs pour la courbe roc
    fpr,tpr,thresholds = metrics.roc_curve(y[idTest],p)
    #ajouter dans la liste sous la forme d'un dictionnaire
    lstRoc.append({'fpr':fpr,'tpr':tpr})

#nombre de couples de valeurs dispo
print(len(lstRoc)) #3, la liste contient 3 couples de vecteurs (fpr, tpr)
```

Nous dessinons les 3 courbes dans un graphique (Figure 1).

```
#graphiques
import matplotlib.pyplot as plt

#les 3 courbes ROC
for v in lstRoc:
    plt.plot(v['fpr'],v['tpr'],lw=1)

#diagonale
plt.plot([0,1],[0,1],linestyle='--',lw=1,color='black')

#légendes et titre
plt.xlabel('False Positive Rate')
plt.ylabel('True Positibve Rate')

#titre
plt.title('[CV] ROC curve for each test fold')

#affichage
plt.show()
```

Il peut y avoir une certaine disparité entre les courbes parce qu'il y a une variabilité inhérente au travail sur les échantillons, mais aussi parce que la taille des blocs, jouant le rôle d'échantillon test à chaque itération, est relativement faible.

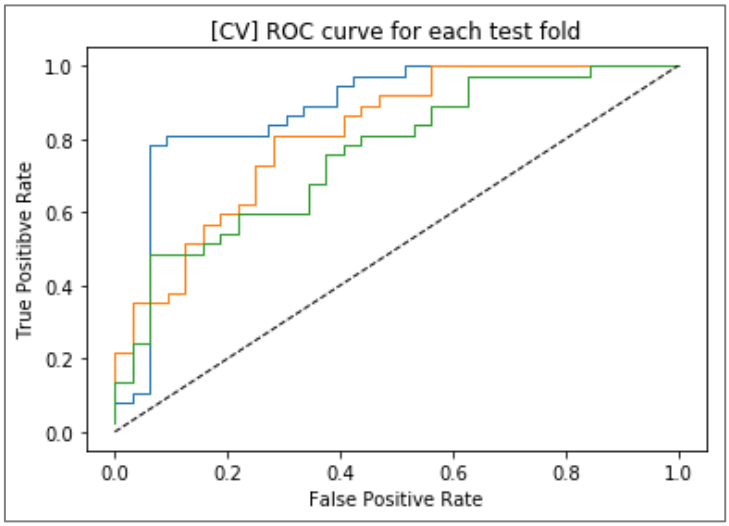

Figure 1 - Courbe ROC pour chaque *fold* de la validation croisée

Pour construire la courbe « moyenne », il s'agirait de prendre les moyennes de *tpr* (ordonnée) pour chaque valeur de *fpr* (abscisse). C'est ici que le bât blesse. On se rend compte en effet que les valeurs de *fpr* sont différentes d'un bloc à l'autre.

Pour le premier,

```
#coordonnées en abscisse courbe n°0
print("FPR(0)",lstRoc[0]['fpr'])
```

```
FPR(0) [0.         0.         0.03030303 0.03030303 0.06060606 0.06060606
 0.09090909 0.09090909 0.27272727 0.27272727 0.3030303  0.3030303
 0.33333333 0.33333333 0.39393939 0.39393939 0.42424242 0.42424242
 0.51515152 0.51515152 1.        ]
```

Le second,

```
#coordonnées en abscisse courbe n°1
print("FPR(1)",lstRoc[1]['fpr'])
```

```
FPR(1) [0.      0.      0.03125 0.03125 0.09375 0.09375 0.125   0.125   0.15625
 0.15625 0.1875  0.1875  0.21875 0.21875 0.25    0.25    0.28125 0.28125
 0.40625 0.40625 0.4375  0.4375  0.46875 0.46875 0.5625  0.5625  1.     ]
```

Le dernier,

```
#coordonnées en abscisse courbe n°2
print("FPR(2)",lstRoc[2]['fpr'])
```

```
FPR(2) [0.      0.      0.03125 0.03125 0.0625  0.0625  0.15625 0.15625 0.1875
 0.1875  0.21875 0.21875 0.34375 0.34375 0.375   0.375   0.40625 0.40625
 0.4375  0.4375  0.53125 0.53125 0.5625  0.5625  0.625   0.625   0.84375
 0.84375 1.     ]
```

Pour dépasser cet écueil, nous définissons un ensemble de 20 points équidistants sur l'abscisse (*fpr*).

```
#suite de 20 points équidistants
mfpr = numpy.linspace(0,1,20)
print(mfpr)
```

```
[0.         0.05263158 0.10526316 0.15789474 0.21052632 0.26315789
 0.31578947 0.36842105 0.42105263 0.47368421 0.52631579 0.57894737
 0.63157895 0.68421053 0.73684211 0.78947368 0.84210526 0.89473684
 0.94736842 1.        ]
```

Et nous reconstituons par interpolation les valeurs de *tpr* (ordonnées) correspondantes pour les 3 courbes ROC.

```
#les 3 suites de valeurs de tpr pour ces valeurs
import scipy

#liste vide
lst_tpr = []

#pour chaque courbe
for curve in lstRoc:
    #estimer par interpolation les valeurs de tpr pour les coordonnées de mfpr
    estimated_tpr = scipy.interp(mfpr,curve['fpr'],curve['tpr'])
    #première valeur en ordonnée = 0
    estimated_tpr[0] = 0.0
    #ajout dans la liste
    lst_tpr.append(estimated_tpr)

#transformation en matrice numpy
mlst_tpr = numpy.array(lst_tpr)
print(mlst_tpr)
```

L'interpolation est réalisée à l'aide de la fonction **interp()** de la librairie « **scipy** ». Nous disposons d'une matrice avec 3 lignes (1 par courbe ROC) et 20 colonnes (pour les 20 points en abscisse).

```
[[0.         0.10810811 0.81081081 0.81081081 0.81081081 0.81081081
  0.86486486 0.89189189 0.94594595 0.97297297 1.         1.
  1.         1.         1.         1.         1.         1.
  1.         1.        ]
 [0.         0.35135135 0.37837838 0.56756757 0.59459459 0.72972973
  0.81081081 0.81081081 0.86486486 0.91891892 0.91891892 1.
  1.         1.         1.         1.         1.         1.
  1.         1.        ]
 [0.         0.24324324 0.48648649 0.51351351 0.54054054 0.59459459
  0.59459459 0.67567568 0.78378378 0.81081081 0.81081081 0.89189189
  0.97297297 0.97297297 0.97297297 0.97297297 0.97297297 1.
  1.         1.        ]]
```

Il ne nous reste plus qu'à calculer les moyennes par colonne pour obtenir les valeurs « moyennes » de *tpr*.

```
#ordonnée moyenne de la courbe ROC
mtpr = numpy.mean(mlst_tpr,axis=0)
print(mtpr)
```

```
[0.         0.23423423 0.55855856 0.63063063 0.64864865 0.71171171
 0.75675676 0.79279279 0.86486486 0.9009009  0.90990991 0.96396396
 0.99099099 0.99099099 0.99099099 0.99099099 0.99099099 1.
 1.         1.        ]
```

Avec (mfpr, mtpr), nous pouvons produire la courbe ROC synthétique de la validation croisée (Figure 2).

```
#plotting
plt.plot(mfpr,mtpr,lw=2,color='y')

#diagonale
plt.plot([0,1],[0,1],linestyle='--',lw=1,color='k')

#légendes
plt.xlabel('False Positive Rate')
plt.ylabel('True Positibve Rate')

#titre
plt.title('[CV] Average ROC curve')
plt.show()
```

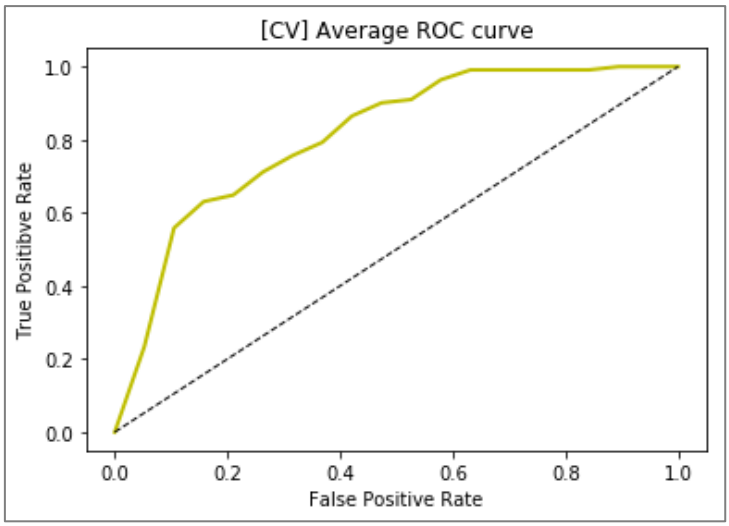

Figure 2 - Courbe ROC synthétique de la validation croisée

Nous pouvons en déduire l'AUC.

```
#AUC de la courbe synthétique
print(metrics.auc(mfpr,mtpr))
```

```
0.8119962067330488
```

C'est une autre interprétation possible du calcul de l'AUC en validation croisée. Le résultat est légèrement différent de l'AUC2 (80.01% ; section 3.3.2) obtenue par moyenne non pondérée des AUC par *fold*.

5 Remarque et conclusion

Par défaut, la descente de gradient stochastique de scikit-learn met à jour les coefficients estimés au passage de chaque observation (online). Pour la stratégie mini-batch où le calcul du gradient s'effectue au passage de groupes (*chunks*) d'observations, nous devons utiliser la méthode partial_fit() de la classe SGDClassifier et la programmer explicitement (MIS, 2014).

Nous avons utilisé une base relativement petite dans ce document. L'idée était de montrer l'utilisation pratique de l'outil et l'influence des paramètres sur la vitesse de convergence et la qualité des modèles qui en découlent. Nos expérimentations menées par ailleurs sur des bases de très forte dimensionnalité (plusieurs dizaines de milliers de descripteurs) montrent que l'outil est particulièrement efficace et véloce. L'enjeu dans ce contexte est de déterminer

les bonnes valeurs des paramètres de régularisation pour prévenir les phénomènes de surapprentissage. Bien ! Voilà un nouveau sujet de cours et de tutoriels en perspective.

6 Références

- (MIS, 2014) Mishra A., « Minibatch learning for large-scale data, using scikit-learn », Adventures in Data Science, December 2014.
- (RAK, 2011) Rakotomalala R., « Pratique de la régression logistique », Juin 2011 ; http://eric.univ-lyon2.fr/~ricco/cours/cours_regression_logistique.html
- (RAK, 2015) Rakotomalala R., « Validation croisée, Bootstrap – Diapos », Février 2015 ; http://tutoriels-data-mining.blogspot.fr/2015/02/validation-croisee-bootstrap-diapos.html
- (RAK, 2018) Rakotomalala R., « Descente de gradient – Diapos », Avril 2018 ; http://tutoriels-data-mining.blogspot.fr/2018/04/descente-de-gradient-diapos.html
- Scikit-learn, « Stochastic Gradient Descent », Section 1.5 ; http://scikit-learn.org/stable/modules/sgd.html
- Scikit-learn, « Receiver Operating Characteristic (ROC) with cross validation », http://scikit-learn.org/stable/auto_examples/model_selection/plot_roc_crossval.html